VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

LADÍCÍ TŘÍDA PRO PHP

DEBUGGING CLASS FOR PHP

BAKALÁŘSKÁ PRÁCE
BACHELOR THESIS

AUTOR PRÁCE **JAKUB KLUVÁNEK**
AUTHOR

VEDOUCÍ PRÁCE **ING. JAN ROUPEC, PH.D.**
SUPERVISOR

BRNO 2011

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2010/2011

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Jakub Kluvánek

který/která studuje v **bakalářském studijním programu**

obor: **Aplikovaná informatika a řízení (3902R001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Ladící třída pro PHP

v anglickém jazyce:

Debugging Class for php

Stručná charakteristika problematiky úkolu:

Třída by měla umožňovat přehledný výpis hodnot proměnných, zachytávat chyby a vyjímky a zobrazovat podrobné informace o tom kde k chybě došlo, tyto chyby logovat a zasílat upozornění emailem. Dále by třída měla umožňovat odhalit pomalá místa v aplikaci.

Cíle bakalářské práce:

Vytvořit snadno použitelnou univerzální třídu, která bude usnadňovat vývoj a ladění aplikací psaných v PHP a umožňovat odhalit pomalá místa v těchto aplikacích.

Seznam odborné literatury:

Andi Gutmans, Stig Saether Bakken, Derick Rethans - Mistrovství v PHP 5
1. vyd. Brno: CP Books, 2005. 655 s. ISBN 80-251-0799-X.
George Schlossnagle - Pokročilé programování v PHP 5
1. vyd. Brno: Zoner Press, 2004. 640 s. ISBN 80-86815-14-5
David Sklar - PHP 5 - moduly, rozšíření a akcelerátory
1. vyd. Brno: Zoner Press, 2005. 341 s. ISBN 80-86815-19-6
Peter Lavin - PHP - Objektově orientované
1.vyd. Praha: Grada Publishing, 2009 224 s. ISBN 978-80-247-2137-8

Vedoucí bakalářské práce: Ing. Jan Roupec, Ph.D.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2010/2011.

V Brně, dne 18.10.2010

L.S.

Ing. Jan Roupec, Ph.D. Ředitel ústavu	prof. RNDr. Miroslav Doupovec, CSc. Děkan fakulty

ABSTRAKT

Cílem této práce je vytvořit pomůcku, která bude pomáhat při vytváření a ladění PHP aplikací. Práce rovněž obsahuje přehled toho, jak se v PHP zachází s chybami a výjimkami standardním způsobem. Výsledkem je snadno použitelná třída, která je snadno rozšiřitelná o další funkčnost a obsahuje všechny základní nástroje užitečné při vývoji. Další, neméně důležitou funkcí, této ladící třídy je část, která zpracovává chyby na produkčním serveru. Tato část je nezbytná, protože uživateli by se neměly zobrazovat detailní informace o chybě. Chyby totiž můžou obsahovat citlivé informace (především hesla do databáze, strukturu tabulek,...). Třídu již sám v praxi používám na několika projektech a ukázala se velmi přínosnou.

ABSTRACT

The aim of this work is to create a tool that assists in developing and debugging PHP applications. It also provides an overview of how PHP handles errors and exceptions in a standard way. The result of this bachelor thesis is easy to use class that is easily extendable for additional functionality and contains all basic tools useful in development. This functions is necessary because users should not view some detailed information about the error. Errors may contain sensitive information (especially passwords for the database, table structure,...). I am using this class in several projects and it seems to be very useful.

KLÍČOVÁ SLOVA

PHP, ladění, chyby, výjimky, zachytávání chyb, zachytávání výjimek

KEYWORDS

PHP, debugging, errors, exceptions, errors handling, exceptions handling

Obsah:

1 ÚVOD

1.1 Co to je ladící třída

Ladící třída by měla především usnadňovat vývoj aplikací. Měla by umět přehledně zobrazovat různé proměnné, zachytávat chyby a výjimky, zobrazovat k nim další informace (aktuální kontext aplikace), případně na ně upozornit emailem. Každý vývojář má své specifické požadavky. Ne všechny lze jednoduše splnit.

Chyby v aplikacích se budou vyskytovat vždy. I kdyby byl kód naprosto dokonalý, vždy může selhat nějaká externí aplikace, nepůjde se připojit do databáze, dojde místo na disku atd. Se všemi těmito chybami musí vývojář počítat. Pokud ošetření některého případu přesto opomene, je žádoucí, aby se o chybě co nejrychleji dozvěděl.

Na druhé straně problému je bezpečnost aplikace. Rozhodně není žádoucí, aby se návštěvníkovi stránky zobrazovala jakákoliv detailní chybová zpráva. Chybové zprávy mohou obsahovat informace o struktuře aplikace, případně i přihlašovací údaje k databázi. Vývojáře aplikace ale takové informace přesto zajímají. Ladící třídy je obvykle řeší tak, že uživateli zobrazí pouhou chybovou zprávu, že došlo k nějaké chybě, a detailní informace o ní někam zapíší. V ideálním případě odešlou vývojáři informativní zprávu například emailem.

Součástí většiny frameworků je nějaká forma ladící třídy. Usnadňuje práci s frameworkem a většinou obsahuje nějakou formu logovací konzole. Většina těchto ladících tříd je však s frameworkem úzce spjata a nelze je použít mimo něj.

1.2 Proč jsem naprogramoval vlastní ladící třídu

Žádná ze současných ladících tříd mi plně nevyhovovala. Především v nich chyběla podpora modifikátorů a možnost zavolat si nějakou funkci při zachycení chyby určitého typu. Proto jsem se rozhodl napsat si ladící třídu sám. V některých jsem se inspiroval a svou ladící třídu naprogramoval tak, aby fungovala podobně. K tomu jsem přidal několik svých vylepšení a celé to zabalil do jedné snadno použitelné třídy. Upravovat kteroukoliv z nich by bylo příliš pracné (snažit se porozumět cizímu kódu,...), ale hlavním mým osobním cílem bylo porozumět lépe chování chyb a výjimek v PHP aplikacích. Pracovně jsem tuto ladící třídu nazval Bug Buster.

2 CHYBY A VÝJIMKY V PHP

V následující části se zmíním o chování chyb a výjimek v PHP. Všechny informace jsou platné pro PHP ve verzi 5. Se staršími verzemi se prakticky už nelze setkat.

2.1 Chyby

Chyby v PHP jsou ve většině případů vyvolány vnitřními funkcemi PHP. Jejich chování ale není příliš sjednocené v rámci všech funkcí.

Chyby se v PHP děli na několik úrovní. Každá úroveň má definovanou svou konstantu, která se dá použít při nastavení zobrazování chyb [1],[3]:

- *E_ERROR* - kritická chyba (nedostatek paměti, nezachycené výjimky,...)
- *E_WARNING* - varování (chybějící argumenty funkce, absence databáze, dělení nulou,...)
- *E_PARSE* - chyba během kompilace (chyba v syntaxi)
- *E_STRICT* - upozornění na možnou dopřednou nekompatibilitu kódu
- *E_NOTICE* - upozornění na možnou chybu (použití nedefinované proměnné,...)
- *E_DEPRECATED* - upozornění na zavrhnuté funkce (většinou z bezpečnostních důvodů)
- *E_CORE_ERROR* - chyba generovaná jádrem PHP
- *E_CORE_WARNING* - varování generované jádrem PHP
- *E_COMPILE_ERROR* - chyba vzniklá během kompilace (podobná *E_PARSE*)
- *E_COMPILE_WARNING* - varování vzniklé během kompilace
- *E_RECOVERABLE_ERROR* - speciální chyba, ze které se dá zotavit (pokud není zachycena error handlerem, je převedena na chybu *E_ERROR*)
- *E_USER_ERROR* - uživatelsky definovaná chyba (způsobí také zastavení běhu skriptu)
- *E_USER_WARNING* - uživatelsky definované varování
- *E_USER_NOTICE* - uživatelsky definované upozornění
- *E_USER_DEPRECATED* - uživatelsky definovaná upozornění na zavrženou funkci
- *E_ALL* - zahrnuje všechny tyto konstanty dohromady s výjimkou *E_STRICT*

Každá z těchto úrovní se chová odlišně. Například chyba při kompilaci způsobí okamžité ukončení skriptu. Takové chyby nelze zachytávat. Rovněž některé chyby (*E_ERROR*) způsobí ukončení skriptu. Některé se dají ale přesto zachytit (použití nedefinované funkce). Chyby typu *E_WARNING* vznikají za běhu skriptu (například otevření neexistujícího souboru) a nezpůsobí ukončení skriptu. Ukončení nezpůsobí ani chyby typu *E_NOTICE* (například použití nedefinované konstanty) nebo použití zavrhnuté funkce - *E_DEPRECATED* (např. funkce pro práci s POSIXovými regulárními výrazy - *ereg*()*).

Notice: Use of undefined constant aa - assumed 'aa' in **C:\wamp\www\tools\errorstest.php** on line **8**

Warning: file_get_contents(abaab) [function.file-get-contents]: failed to open stream: No such file or directory in **C:\wamp\www\tools\errorstest.php** on line **5**

Fatal error: Call to undefined function abc() in **C:\wamp\www\tools\errorstest.php** on line **6**

Parse error: syntax error, unexpected T_STRING in **C:\wamp\www\tools\errorstest.php** on line **7**

Deprecated: Function ereg() is deprecated in **C:\wamp\www\tools\errorstest.php** on line **8**

Obr. 1 Chyby úrovně E_NOTICE, E_WARNING, E_ERROR, E_PARSE a E_DEPRECATED v PHP (bez rozšíření)

Hlavní nevýhodou chyb oproti výjimkám je to, že se dají snadno ignorovat. Stačí k tomu pouhé nastavení konfigurační direktivy *display_errors* na *off*. Poté se nebudou zobrazovat žádná chybová hlášení. Pokud však dojde k chybě, která ukončí běh skriptu, nezobrazí se pravděpodobně

vůbec nic (záleží na tom, zda webový server již zaslal prohlížeči nějaká data).

Další možností filtrování chyb je použití funkce *error_reporting()* (je to alternativa ke konfigurační direktivě *error_reporting*), pomocí které lze nastavit, které úrovně chyb se budou zobrazovat. Jako hodnotu lze předat i *E_ALL*, což znamená všechny úrovně chyb.

Chyby můžeme vytvářet také ručně. Pro tyto účely jsou zavedeny další úrovně *E_USER_ERROR*, *E_USER_WARNING*, *E_USER_NOTICE*, *E_USER_DEPRECATED*. K tomu slouží funkce *trigger_error()*, kterou když zavoláme, tak je vyvolána chyba zvolené úrovně se zadanou chybovou zprávou.

2.1.1 Zachytávání chyb

Chyby lze v PHP zachytávat v zásadě dvěma způsoby. Buď přímo funkcí na zachytávání chyb (error handlerem), nebo pomocí funkce, která se volá při ukončení skriptu (shutdown handlerem).

Error handler se nastavuje zavoláním funkce *set_error_handler()*. Jako parametr slouží název funkce, která se zavolá při vyvolání chyby. Této volané funkci se jako parametry předává kód chyby, její popis a název souboru s řádkem, na kterém došlo k chybě. Tímto způsobem však nelze zachytávat chyby úrovně *E_ERROR* a všech dalších, které způsobí ukončení skriptu.

Pomocí druhé možnosti (shutdown handleru) lze zachytit i některé chyby úrovně *E_ERROR*. Jako příklad lze použít volání nedefinované funkce. V shutdown handleru si pomocí funkce *error_get_last()* zjistíme, zda na stránce došlo k chybě. Poté detaily o této chybě můžeme opět zpracovat pomocí stejné funkce jako v předchozím případě.

2.2 Výjimky

Výjimky tvoří nedílnou část moderního programování v PHP. Již málokdo tvoří v PHP bez použití objektů. A právě zde přichází na řadu výjimky.

Výjimka samotná je objekt, který má svůj název a několik metod na získání chybové zprávy atd. Pro správné chování musí mít výjimky společného předka - výjimku *Exception*. Proto pokud chceme definovat výjimku vlastní, tak ji musíme od této třídy také odvodit. Výjimky lze vyvolat (vyhodit) tak, že v kódu uvedeme:

```
throw new Exception('Exception message');
```

Př 1. Vyvolání výjimky

Pokud tak učiníme, objeví se nám zpráva podobná chybě (Obr. 2). Chybová zpráva je navíc obohacena o tzv. backtrace. Backtrace je seznam aktuálních aktivních volání funkcí v daném skriptu. Například když funkce *a()* zavolá funkci *b()*, tak v backtrace najdeme obě tyto funkce. Backtrace většinou obsahuje i informace o předaných parametrech té dané funkci.

```
Fatal error:  Uncaught exception 'Exception' with message 'Exception message' in C:\wamp\www\kDebug\example-exception-full.php:11
Stack trace:
#0 C:\wamp\www\kDebug\example-exception-full.php(14): testException('aa', 'bb')
#1 C:\wamp\www\kDebug\index.php(141): require_once('C:\wamp\www\kDe...')
#2 {main}
  thrown in C:\wamp\www\kDebug\example-exception-full.php on line 11
```

Obr. 2 Nezachycená výjimka v PHP (bez rozšíření)

Výjimky mají oproti chybám tu výhodu, že je nelze nijak jednoduše ignorovat pouhým nastavením nějaké konfigurační direktivy. Vždy je třeba se s nimi vypořádat přímo v kódu. Nejsprávnější postup je použití dvojice *try-catch*, která je obdobná jako například v jazycích C.

```
$array = array(NULL, 0, 3, 4, 5);
$a = $array[rand(0, 4)];
try
{
	if($a === NULL)
		throw new InvalidArgumentException('Parametr nesmí být NULL');
	if($a == 0)
```

```
            throw new Exception('Dělení nulou!');
      $b = 300 / $a;
      echo 'Konec bloku try.<br />';
}
catch(InvalidArgumentException $e)
{
      echo 'Výjimka: '.$e->getMessage();
      die;
}
catch(Exception $e)
{
      echo 'Zachycená výjimka: '.$e->getMessage().'<br />';
      $b = 6;
}
echo '$b = '.$b;
```

Př 2. Zachycení různých výjimek

Tento kód bude náhodně generovat různé výjimky nebo jeho vykonání projde v pořádku. V kódu jsou vidět dvě hlavní části. První je část *try*, druhá část *catch*. V části *try* se provádí kód, který může vyvolat výjimku (i volání nějaké funkce, která výjimku vyvolává). V příkladu jsou použity dvě různé výjimky pro demonstraci možností zachytávání výjimek. To se děje v sekcích *catch*. Každá z nich je určena jednomu typu výjimek. Při zachytávání výjimek se používá principu dědičnosti tříd. Pokud bychom tedy vynechali v příkladu část, kde se zachycuje výjimka *InvalidArgumentException*, byla by zachycena v druhém bloku catch, protože výjimka *InvalidArgumentException* dědí z výjimky *Exception*. Po zpracování kódu a případném zachycení výjimky skript pokračuje dále v činnosti (pokud v části catch neukončíme skript například příkazem *die*). Výhodou je, že nemusíme ošetřovat každou chybu zvlášť, ale můžeme to udělat až pohromadě.

Skript navíc může při výskytu nějaké nefatální chyby (např. nenačtení počítadla) pokračovat dál v činnosti.

Pokud nezachytíme výjimky v bloku *catch*, dochází k chybě, kdy je nezachycená výjimka zobrazena (Obr. 2). To je způsobeno buď chybnou logikou aplikace, nebo leností programátora, který nepochopil výjimky a používá je stejně jako chyby.

Velkou nevýhodou PHP je to, že ho nelze nijak přepnout tak, aby vestavěné funkce vyvolávaly výjimky místo chyb (alespoň v současné verzi PHP 5 toto nelze provést). O to se musíme postarat sami.

2.2.1 Probublávání výjimek

K probublávání výjimky dochází v případě, kdy ji ve funkci nezachytíme.

```
function funkce1() { throw new InvalidArgumentException('Text chyby'); }
function funkce2()
{
      try
      {
            funkce1();
      }
      catch(IOException $e)
      {
            die('IO Exception');
      }
}

try
{
      funkce2();
}
catch(Exception $e)
```

```
{
	die($e->getMessage());
}
```

Př 3. Probublávání výjimky

V ukázce je vidět, že *funkce1()* zachytávání neřeší vůbec. Výjimka tedy postoupí do volající *funkce2()*. V ní se sice zachycení výjimky řeší, ale nezachytí se výjimka *InvalidArgumentException*. Proto se postoupí ještě do vyšší úrovně (samotného skriptu), kde se již výjimka zachytí. Pokud by ani na této úrovni nedošlo k zachycení výjimky, vznikne nezachycená výjimka, což způsobí chybu úrovně *E_ERROR* a následné ukončení běhu skriptu.

2.2.2 Zachycení nezachycených výjimek

U výjimek je jejich zachytávání podstatně snažší, než je tomu u chyb. Podobně jako u chyb máme k dispozici exception handler. Nastavíme ho zavoláním funkce *set_exception_handler()*. Poté jsou všechny nezachycené výjimky předány naší funkci a je jen na ní, jak s nimi bude zacházet. Při detekci první nezachycené výjimky (i když je zachycena exception handlerem) dochází k ukončení běhu skriptu podobně jako je tomu u chyby úrovně *E_ERROR*.

2.3 Alternativní způsoby práce s chybami

V dřívějších dobách (u verzí 4 a starší) se pracovalo s chybami tak, že se vytvořil cyklus *do-while*, který proběhl pouze jednou. V případě výskytu chyby se z cyklu poté dá přejít do bezpečné části (za cyklus) pomocí příkazu *break*. Řešení je to podobné dvojici *try-catch*. Alternativou k Př 2. může tedy být následující řešení:

```
$array = array(NULL, 0, 3, 4, 5);
$a = $array[rand(0, 4)];
$error = NULL;
$errorText = '';
do
{
	if($a === NULL)
	{
		$error = 'invalid_argument';
		$errorText = 'Parametr nesmí být NULL';
		break;
	}
	if($a == 0)
	{
		$error = 'division_zero';
		$errorText = 'Dělení nulou!';
		break;
	}
	$b = 300 / $a;
	echo 'Konec bloku do..while.<br />';
}
while(false);

if($error == 'invalid_argument')
{
	echo 'Chyba: '.$errorText;
	die;
}
elseif($error == 'division_zero')
{
	echo 'Dělení: '.$errorText;
	die;
```

```
}
echo '$b = '.$b;
```

Př 4. Práce s chybami v cyklu do-while

2.4 Konstrukce or xxx

Velice zajímavou a netradiční konstrukcí pro ošetřování chyb je konstrukce *or xxx* [2].

```
$a = mysql_query('SELECT * FROM tabulka')
     or die('Chyba v SQL.');
```

Př 5. Použití konstrukce or die();

Pokud je výraz před klíčovým slovem *or* vyhodnocen jako false, je zavolán výraz následující. Nejčastěji klíčové slov *or* následuje funkce *die()*. Ale lze ji nahradit za jakoukoliv jinou.

```
$a = mysql_query('SELECT * FROM tabulka')
     or print('Chyba v SQL');
```

Př 6. Použití konstrukce or print();

2.5 Ignorování chyb (scream operátor)

Chyby je v PHP možné i ignorovat. Slouží k tomu operátor @ (scream). Pokud ho uvedeme před voláním funkce, tak je chyba generovaná tou funkcí ignorována. Ignorovány jsou jak vestavěné chyby, tak ty uživatelsky generované. Lze jej však použít pouze pro potlačení zobrazení chyby. Pokud dojde k chybě, která ukončí běh skriptu, tak dojde k jeho ukončení i při použití tohoto operátoru. Error handler bude zavolán i když je použit scream operátor. Odhalit použití scream operátoru v error handleru je možné díky tomu, že při jeho použití je dočasně snížena úroveň oznamování chyb na nulu (*error_reporting() == 0*)

```
$a = @file_get_contents('unknownfile.txt');
```

Př 7. Použití scream operátoru

3 STÁVAJÍCÍ ŘEŠENÍ

3.1 Xdebug

Xdebug je rozšíření PHP, které slouží pro ladění PHP aplikací. Ne vždy je však k dispozici a jeho instalace je někdy složitá. Po jeho instalaci začne sám obsluhovat všechny chybové hlášky. Velkou výhodou Xdebugu je jeho vysoká konfigurovatelnost [6]. Téměř vše se dá nastavit v souboru *php.ini* (což může být nevýhoda, pokud k němu nemáme přístup a funkce *ini_set()* je zakázaná).

(!) Notice: Use of undefined constant aa - assumed 'aa' in C:\wamp\www\tools\errorstest.php on line *9*

Call Stack

#	Time	Memory	Function	Location
1	0.0005	675776	{main}( )	..\errorstest.php:0

(!) Warning: file_get_contents(abaab) [function.file-get-contents]: failed to open stream: No such file or directory in C:\wamp\www\tools\errorstest.php on line *5*

Call Stack

#	Time	Memory	Function	Location
1	0.0003	676176	{main}( )	..\errorstest.php:0
2	0.0004	676560	file_get_contents ( )	..\errorstest.php:5

(!) Fatal error: Call to undefined function abc() in C:\wamp\www\tools\errorstest.php on line *6*

Call Stack

#	Time	Memory	Function	Location
1	0.0003	676168	{main}( )	..\errorstest.php:0

(!) Parse error: syntax error, unexpected T_STRING in C:\wamp\www\tools\errorstest.php on line *7*

(!) Deprecated: Function ereg() is deprecated in C:\wamp\www\tools\errorstest.php on line *8*

Call Stack

#	Time	Memory	Function	Location
1	0.0007	676032	{main}( )	..\errorstest.php:0

Obr. 3 Chyby úrovně E_NOTICE, E_WARNING, E_ERROR, E_PARSE a E_DEPRECATED v PHP s použitím rozšíření Xdebug

Na Obr. 3 je vidět, že většina chybových hlášek obsahuje i backtrace. V závislosti na konfiguraci Xdebugu si zde můžeme nechat vypsat i vybrané superglobální proměnné, případně proměnné v aktuálním kontextu chyby.

Fatal error: Uncaught exception 'Exception' with message 'Exception message' in C:\wamp\www\kDebug\example-exception-full.php on line 21

Exception: Exception message in C:\wamp\www\kDebug\example-exception-full.php on line 21

Call Stack

#	Time	Memory	Function	Location
1	0.0021	711976	{main}( )	..\index.php:0
2	0.0336	818736	require_once('C:\wamp\www\kDebug\example-exception-full.php')	..\index.php:141
3	0.0336	819576	testException($param1 = 'aa', $par2 = 'bb')	..\example-exception-full.php:23

Dump $_SERVER

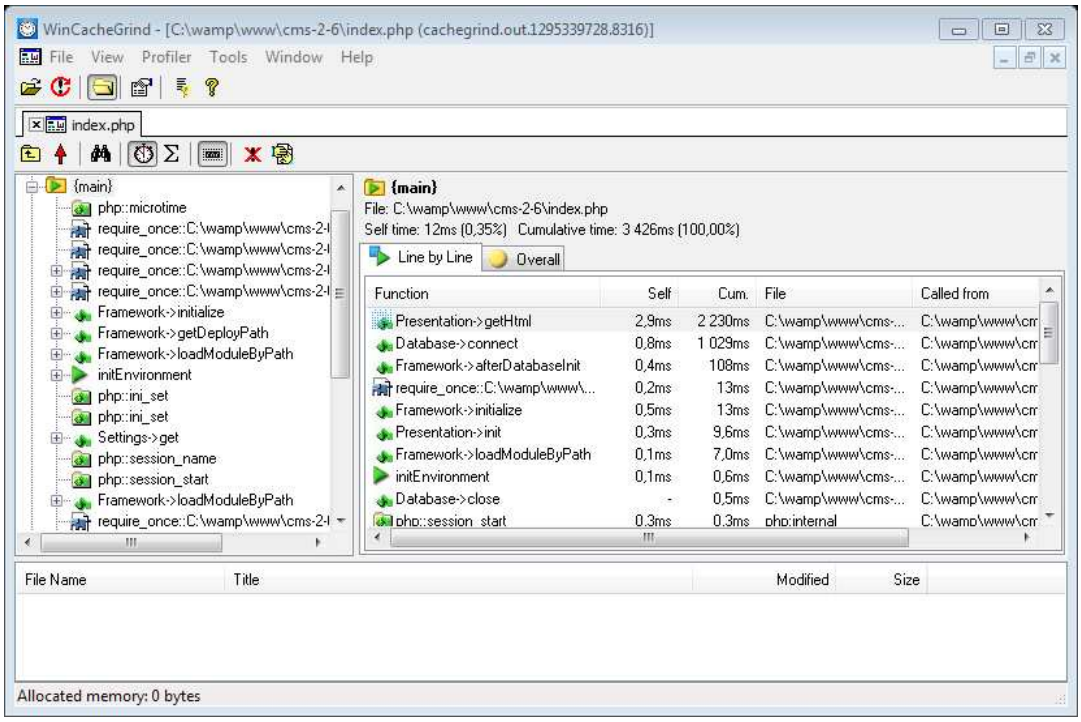
$_SERVER['REQUEST_URI'] =	string '/kDebug/?example=exception-full' (length=31)

Variables in local scope (#3)

$c =	int 3
$par2 =	string 'bb' (length=2)
$param1 =	string 'aa' (length=2)

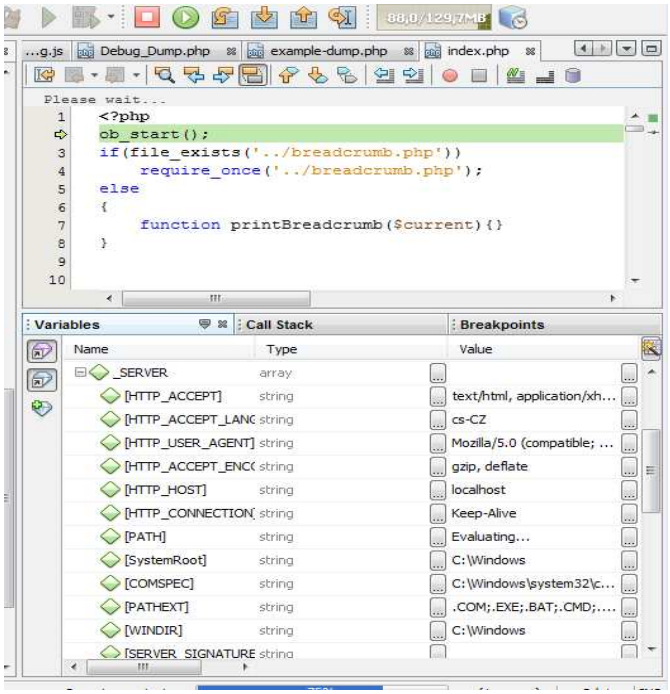
Obr. 4 Nezachycená výjimka v PHP s rozšířením Xdebug

Xdebug obsahuje také profiler, který zapisuje data do souboru. Tento soubor poté můžeme analyzovat pomocí nějakého nástroje (například WinCacheGrind) a odhalit tak nejpomalejší místa v aplikaci.



Obr. 5 WinCacheGrind s načteným výpisem z Xdebugu

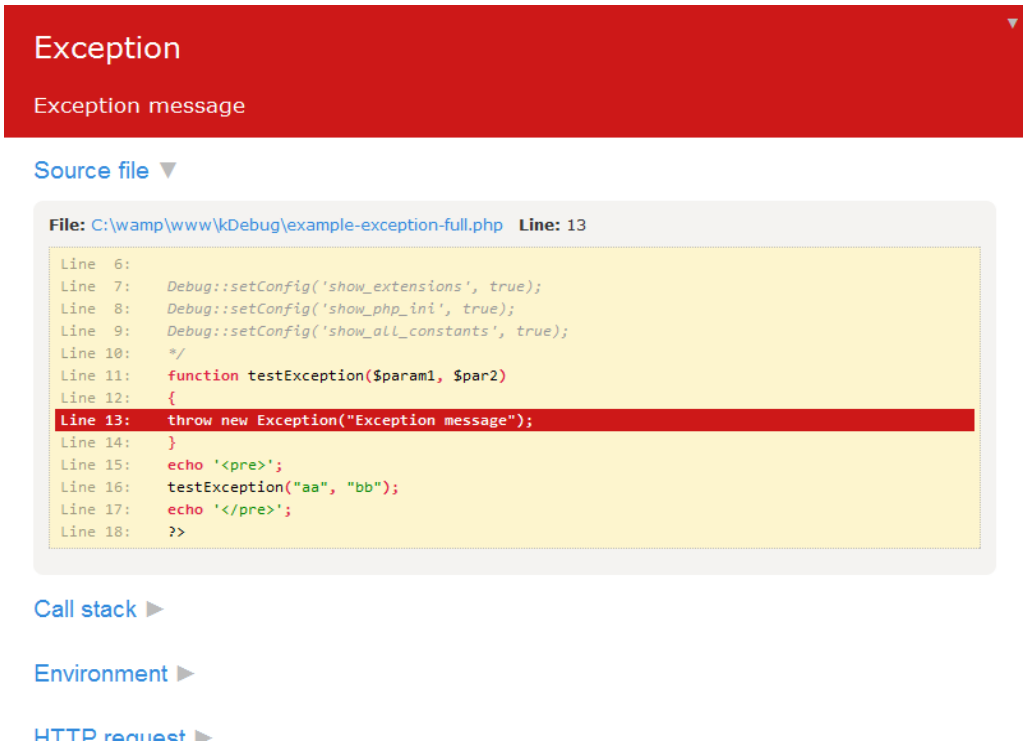
Xdebug lze rovněž připojit na některá vývojová prostředí, která mají jeho podporu implementovánu (Netbeans, Eclipse,...). Poté lze v těchto prostředích nastavit breakpointy (místa, na kterých se vykonávání kódu zastaví, dokud není vývojářem povoleno pokračování) a kód krokovat stejně jako programy např. v jazycích C.



Obr. 6 Ladění aplikací v Netbeans s pomocí Xdebugu

3.2 Nette\Diagnostics\Debugger (Laděnka)

Nette framework je známý PHP framework od českého programátora Davida Grudla [5]. Součástí tohoto frameworku je také ladící třída - Laděnka. Laděnka byla tím základním impulsem a největší inspirací pro tvoru vlastní ladící třídy. Obsahuje spoustu zajímavých a užitečných funkcí. Od zachytávání chyb a výjimek (a upozorňování na ně emailem) až po jednoduché stopky, výpis proměnných,... Součástí Laděnky je rovněž profiler, který je použitelný s Nette frameworkem. Tento framework nepoužívám, proto jsem neměl možnost tento profiler vyzkoušet.



Obr. 7 Výjimka v Laděnce

Na Obr. 7 je vidět ukázka zachycené výjimky právě pomocí Laděnky. Obsahuje jak základní informace o chybě/výjimce (název + popisek) tak detailnější informace o místu vzniku chyby (backtrace, hlavičky dotazu a odpovědi, superglobální proměnné,...).

3.3 PHPDebug

PHPDebug funguje na odlišném principu než Laděnka. Na každé stránce vypíše malý panel v pravém horním rohu. Po kliknutí na některou z jeho záložek se zobrazí více informací o dané sekci. Vypsat lze rovněž superglobální proměnné, dokáže odeslat html kód k validaci a obsahuje log konzoli. Do ní lze zapisovat z PHP jakékoliv zprávy. Ke zprávě je připojen název souboru a řádek, ze kterého se logování provedlo.

Obr. 8 Log konzole v PHPDebugu

4 JÁDRO LADÍCÍ TŘÍDY

Mým původním cílem bylo vytvořit ladící třídu tak, aby byla v jednom souboru. Postupem času se ale třída zvětšovala a bylo třeba ji rozdělit na několik podtříd. Přesto se ale povedlo zachovat stejný přístup k jednomu statickému objektu s názvem *Debug*. Objekt je statický, proto mu poté není třeba vytvářet instanci (ve skutečnosti se objektu vytvoří instance při spuštění PHP skriptu, která je platná až do jeho ukončení). Navíc je potom dostupný i v shutdown handleru, kde bychom museli jinak jeho instanci při výskytu chyby znovu vytvářet.

Rozhodl jsem se pro řešení, které by zachovalo stávající API třídy a přitom umožňovalo neomezené rozšiřování do budoucnosti pouze s jednoduchou úpravou. Základem je třída Debug, která má metodu *__callStatic()*, ta teprve volá patřičné podtřídy. To, která metoda je v které podtřídě zařazena určuje pole, které má následující strukturu:

```
$subClasses = array('SubClassName' => array('method1', 'met2'));
```

Př 8. Struktura podtříd

Tento způsob má několik velkých výhod. Největší výhodou je to, že i když je třída rozdělená do několika podtříd, které spolu zdánlivě nijak nesouvisí, tak se všechny chovají jako metody objektu *Debug*. Některé podobjekty lze tedy použít i samostatně bez ostatních podtříd.

Další výhodou je ta vlastnost, že celou třídu lze snadno vypnout. Proto jsem zavedl proměnnou *$isOn*, která má výchozí hodnotu *true*. Pokud se nastaví na *false*, metoda *__callStatic()*, která se stará o volání metod v podtřídách, bude vracet pouze hodnotu *NULL* a nebude je tedy volat.

Zajímavou výhodou je možnost třídu rozšiřovat o další metody pouhým vytvořením funkce se správným názvem. Pokud totiž zavoláme na objektu *Debug* metodu *nedefinovanaMetoda()*, zjistí se, jestli existuje funkce s názvem *kdebug_prototype_nedefinovanaMetoda()* a pokud ano, tak se zavolá. Z tohoto důvodu jsou také téměř všechny metody i proměnné v podobjektech veřejné (aby se mohly volat z těchto rozšiřujících funkcí).

4.1 Šablonový systém

Postupem času vznikla potřeba oddělit HTML kód od PHP kódu. K tomu je obvykle v praxi nejsnazší použít nějaký hotový šablonovací systém (například Smarty nebo Dwoo). Ladící třídu jsem však nechtěl stavět na něčem, co je tak komplexní jako například Smarty. Většinu funkcí bych v celém projektu nevyužil. Vytvořil jsem proto velice jednoduchý šablonovací systém. Celý je obsažený v podtřídě *Debug_Tpl* a vejde se do pouhých 99 řádků kódu. Jeho jádrem jsou 2 hlavní metody. První s názvem *assign()* přiřadí do šablony proměnnou. Druhá s názvem *fetch()* vrátí zpracovaný obsah šablony. Metoda *fetch()* nejdříve projde všechny přiřazené proměnné a přichystá je do nacachované verze šablony. Poté v šabloně nahradí složené závorky za otevírací resp. uzavírací tagy PHP (*<?php*, *?>*) a dvojici *{$* za *<?php echo*. Tím je celá nacachovaná verze šablony hotová. Tento obsah poté uložíme do nějakého souboru a za pomocí output bufferu získáme obsah, který by se normálně odeslal do prohlížeče a vrátíme ho z metody *fetch()* ven. Tento systém je tedy maximálně jednoduchý. Šel by například vylepšit o trvalejší cachování šablon. V tomto případě je to ale řešení zcela dostačující.

```
<div class="Tests">
      {foreach($test as $test):}
            {if($test != 3)}
                  {$test}<br />
            {endif;}
      {endforeach;}
</div>
```

Př 9. Ukázka šablonovacího systému

4.2 Sloučení do jednoho souboru

Použitím podtříd a šablon se narušila původní myšlenka mít vše v jednom souboru. Vytvořil jsem tedy jednoduchý skript, který vše opět sloučí dohromady. U samotných definicí tříd stačí pouhé načtení obsahu souboru s podtřídou a vložení na konec společného souboru. Se šablonami to ale tak jednoduché není. V jednosouborové verzi je proto zavedena globální proměnná typu pole *$DebugSuperVariables*. Jako klíč je použit název souboru se šablonou a jako hodnota jeho obsah. Třída *Debug_Tpl* je uzpůsobená tak, aby se s takovou šablonou dovedla vypořádat. Podobným způsobem se vkládá také soubor s JavaScriptem, CSS styly, konfigurační soubor Debug.ini a pomocné třídy Mailer a String.

4.3 Produkční a vývojové prostředí

BugBuster obsahuje možnost nastavit, zda se nachází v produkčním či vývojovém prostředí. Na produkčním serveru je logické, že by se návštěvníkovi neměly zobrazovat ladící informace. Může se tak klidně stát, že například zapomeneme v kódu volání metody *Debug::dump($variable).* Na produkčním serveru se však tyto informace vůbec nezobrazí. Pokud je třeba přesto obejít toto výchozí chování metody *dump()*, stačí místo ní zavolat metodu *prod_dump()*, která příjmá stejné parametry, ale nebere ohled na nastavení produkčního/vývojového prostředí. Podobné alternativy potom existují k většině metod, které něco zobrazují v prohlížeči.

4.4 Javascript

Pro usnadnění a urychlení vývoje je v BugBusteru použita JavaScriptová knihovna jQuery a jQuery UI. Nevýhodou je, že pro plný komfort musí být na každé stránce vložena knihovna jQuery (jQuery UI je použita pouze pro plovoucí panel). V konfiguraci se však dá nastavit, aby se tyto knihovny vůbec nenahrávaly (například se o jejich nahrání postaráme sami). Celý BugBuster funguje tedy i s vypnutým JavaScriptem. Neposkytuje však takový komfort při používání.

5 BUG BUSTER

5.1 Výpis proměnných

Výpis obsahu proměnných je nejčastěji používaná součást ladění aplikací. Realizovat se dá různými způsoby. Nejjednodušší je použití příkazu *echo*.

```
$var = 'Obsah proměnné.';
echo $var;
```

Př 10. Výpis proměnné pomocí příkazu echo

Takto ovšem můžeme vypisovat pouze textové proměnné nebo čísla (případně objekty, které mají definovánu metodu *__toString()*). Pokud chceme vypsat proměnnou libovolného datového typu, můžeme použít funkci *var_dump()*. Ta vypíše datový typ proměnné a případně i délku řetězce.

```
array
  0 => int 1305389238
  1 => string '10:00:12 1.2.2009' (length=17)
  2 =>
    array
      0 => string 'a' (length=1)
      1 => string 'b' (length=1)
  3 => string 'foo/bar' (length=7)
```

Obr. 9 Výpis pole funkcí var_dump() se zapnutým Xdebugem

Tyto informace jsou v mnoha případech dostačující. Někdy ale například ukládáme do databáze datum ve formátu timestamp a potřebujeme si ho kvůli kontrole převést na nějaký čitelný formát. K tomu můžeme buď použít nějakou online službu nebo využít možností BugBusteru, který k proměnným přidává modifikátory.

Obr. 10 Výpis proměnných v BugBusteru s modifikátory

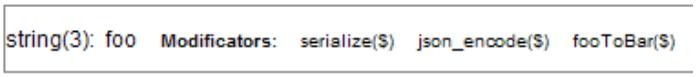
Modifikátory jsou při prvním zobrazení skryty. Teprve po kliknutí na text "Modificators" se zobrazí dostupné modifikátory pro danou hodnotu. Po kliknutí na některý z nich se zobrazí jeho hodnota (hodnota proměnné převedená nějakou funkcí). Na Obr. 10 je vidět zobrazený modifikátor, který převedl číslo na datum. Modifikátory se zobrazují v závislosti na tom, jakou hodnotu má proměnná. Nezobrazují se tedy vždy ty stejné. Modifikátory lze také jednoduše přidávat.

```
function myOwnModificator($variable)
{
	$variable = (string)$variable;
	if($variable != 'foo')
		return '';
	$value = 'bar';
	return Debug_Modificators::printModificator('fooToBar($)', $value);
```

```
}
Debug::registerModificator('foo_to_bar', 'myOwnModificator');
$foo = 'foo';
Debug::dump($foo);
```

Př 11. Přidání vlastního modifikátoru

Nejdříve vytvoříme funkci (*myOwnModificator()*), která bude představovat samotný modifikátor a jako jediný parametr příjmat hodnotu proměnné. Pokud tato funkce vrátí prázdný řetězec, modifikátor se nevypíše. Modifikátor musí vracet obsah vrácený metodou *printModificator()*. Jako první parametr jí předáme text, který se zobrazí při výpisu proměnné a jako druhý parametr upravenou hodnotu. Nově vytvořený modifikátor poté zaregistrujeme zavoláním metody *registerModificator()*. Tato metoda jako první parametr příjmá název modifikátoru (který se poté dá použít k odregistrování tohoto modifikátoru) a jako druhý parametr název funkce, která představuje samotný modifikátor.



Obr. 11 Výpis vlastního modifikátoru

Velkou nevýhodou použití originální funkce *var_dump()* je to, že výsledek rovnou zobrazí do prohlížeče. To není v některých případech žádoucí (například pokud chceme výpis pouze zaznamenat někam do databáze). BugBuster proto má metody *returnDump()* a *dumpToFile()*. První metoda výpis proměnné vrátí místo přímého vypsání a druhá tento obsah rovnou zapíše do zadaného souboru.

5.1.1 Firebug

Pravděpodobně nejužitečnějším rozšířením se kterým se může vývojář potkat je Firebug v prohlížeči Firefox. Umožňuje prozkoumávat jednoduše HTML kód. Do Firebugu existuje rozšíření FirePHP, které umožňuje z PHP skriptů odesílat do Firebugu obsah nějaké proměnné. Zprávy se do prohlížeče posílají prostřednictvím HTTP hlaviček s danou strukturou [4]. Výhodou je tedy to, že vypisovat obsah proměnných tak můžeme částečně i na produkčním serveru. Tento postup je však vhodný pouze jako krajní řešení (lepší je výpis například uložit do souboru a poté si ho stáhnout). Pokud návštěvník nemá nainstalovaná rozšíření, tak si zprávu nedokáže přečíst. Hlavní výhodou tohoto postupu je to, že stránka není zahlcena výpisem proměnné, ale ta je vypsána ve vedlejším okně nebo do logové konzole.

Od vývojářů rozšíření FirePHP existuje knihovna, která s tímto rozšířením dokáže komunikovat. Je však příliš obsáhlá a obsahuje vlastní error a exception handlery. Proto jsem se rozhodl tento protokol implementovat sám. Samotné odeslání zprávy se poté vešlo do pouhé jedné krátké metody.

V ostatních prohlížečích prozatím není dostupné žádné podobné rozšíření, které by dokázalo plně nahradit Firebug+FirePHP. Pokud už nějaké existuje, jeho funkčnost není ideální a nedá se na něj spolehnout. Jako příklad může posloužit rozšíření PhpConsole do prohlížeče Google Chrome. Rozšíření však používá pro přenos dat cookies, což je značně limitující (velikost i počet cookies je v prohlížečích omezena). Při vývoji této části BugBusteru na to bylo přesto potřeba myslet pro případ, že by se nějaké použitelné rozšíření v budoucnu objevilo. API této části třídy proto vychází právě z vlastností rozšíření FirePHP.

```
Debug::browserLog($_SERVER, 'volitelný popisek');
Debug::browserInfo('Nějaká informace');
Debug::browserWarn('Varování');
Debug::browserError('Chyba!');
Debug::browserTable(array(array(0,1), array(1,2)));
Debug::browserGroupStart('název skupiny');
Debug::browserError('Chyba uvnitř skupiny!');
Debug::browserGroupEnd();
```

```
Debug::browserTrace();
Debug::browserException(new Exception());
```

Př 12. Ukázky použití práce s Firebugem

Metody *browserLog/Info/Warn/Error* jsou prakticky shodné. Jediný rozdíl mezi nimi je v ikonce, která se u zprávy zobrazuje. Metoda *browserTable()* zobrazí přehlednou tabulku ze zadaných hodnot. První řádek se použije jako záhlaví a vypíše se tučně. Metoda *browserGroupStart()* zahajuje skupinu. Další odeslané zprávy budou zařazeny do této skupiny. Skupina se poté musí ukončit metodou *browserGroupEnd()*. Backtrace k aktuálnímu místu zapíšeme metodou *browserTrace()*. Výjimku a veškeré standardní informace o ní do prohlížeče pošleme zavoláním metody *browserException()*.

Třída je vnitřně implementována tak, že pouze volá podobné metody jiného objektu. Toto řešení jsem zvolil proto, že pokud se objeví nějaké jiné použitelné rozšíření, implementuje se do zvláštního objektu a v metodách *browser*()* se budou pouze volat metody toho objektu stejně jako je tomu v případě FirePHP.

5.2 Zpracování chyb a výjimek

O chování chyb a výjimek jsem se již zmínil v kapitole2. Chyby a výjimky se v BugBusteru zachytávají standardním způsobem pomocí error a exception handleru. Chyby, které nelze tímto jednoduchým způsobem zachytit (*E_ERROR*,...) se zkouší zachytit v shutdown handleru, který poté zavolá stejné metody jako error handler (chyby potom mají jednotný vzhled).

5.2.1 Chyby

Práce s chybami je většinou velice jednoduchá. Většina chyb, které v aplikaci vzniknou, nejsou fatální. Error handler má proto celkem lehkou práci. Zjistí si pouze název chyby a poté vypíše další užitečné informace k chybě (všechny tyto části lze pomocí konfigurace BugBusteru zakázat):

- zdrojový kód kolem místa vzniku chyby (se zvýrazněním syntaxe)
- backtrace od místa vzniku chyby
- superglobální proměnné (*$_GET, $_POST, $_SERVER*,...)
- definované konstanty
- vložené soubory
- zapnutá php rozšíření
- konfigurační hodnoty z php.ini
- HTTP hlavičky dotazu a odpovědi

Obr. 12 Chyba zachycená BugBusterem

5.2.2 Akce při chybě

Zajímavou vlastností BugBusteru je to, že lze v konfiguraci nastavit funkce, které se provedou ještě před samotným zpracováním chyby. Lze si tak nastavit například vlastní logování.

```
Debug::addConfig('on_notice', 'testFunction');
```

Př 13. Nastavení zavolání funkce testFunction pri chybě úrovně E_NOTICE

Nastavení můžeme provádět s hodnotami:

- on_warning
- on_fatal_error
- on_parse_error
- on_notice
- on_strict
- on_deprecated
- on_recoverable_error
- on_unknown_error

Jako druhý parametr se vždy předává název funkce, která se bude volat. Této funkci jsou parametry předány kód chyby, její název, soubor a řádek, kde k chybě došlo.

5.2.3 Přeložení chyby na výjimku

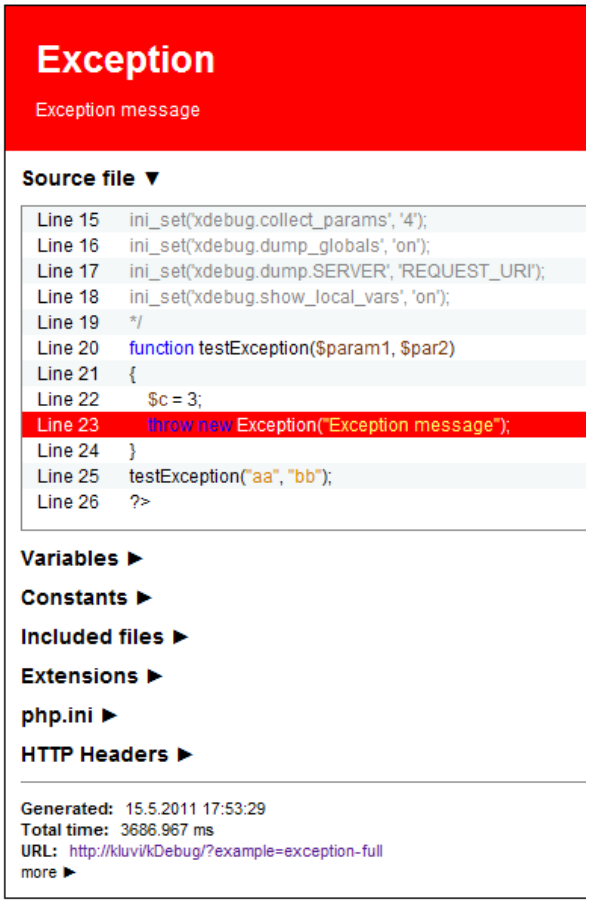
Chybu lze také volitelně převést na výjimku. K tomu je již v PHP nachystaná výjimka *ErrorException*. BugBuster tedy v případě, že je to nastaveno v konfiguraci, fatální chybu převede na tento typ výjimky.

5.2.4 Výjimky

Samotný výpis výjimek vypadá podobně jako u chyb. Narozdíl od chyb však musíme

funkčnost rozšířit, protože se výjimky dají zachytávat v bloku *catch*. Všem metodám se jako jediný parametr předává objekt s výjimkou.

Metoda *getExceptionHtml()* pouze vrátí HTML kód k dané výjimce. Neprovádí se zde žádné akce. Metoda *showException()* výjimku zobrazí (v závislosti na nastavení produkčního módu zobrazí/nezobrazí detailní informace). Metoda *logException()* ten stejný kód co by se zobrazil, zaznamená do souboru. Nejobsáhlejší je metoda *processException()*, která výjimku odešle metodě *browserException* (odešle do Firebugu), zavolá funkce, které se mají vykonat při dané výjimce (podobně jako on_notice,...), výjimku zaloguje a vrátí HTML kód, který si potom můžeme zobrazit nebo uložit.



Obr. 13 Výjimka zachycená BugBusterem

5.2.5 Akce při výjimce

Při zachycení výjimky lze rovněž vyvolat nějakou akci podobně jako při chybě.

```
Debug::addConfig('on_exception', 'testFunction',
'InvalidArgumentException');
```

Př 14. Nastavení akce při výjimce InvalidArgumentException

Tímto nastavením se při každé výjimce *InvalidArgumentException* zavolá funkce *testFunction()*. Poslední parametr je volitelný. Pokud ho neuvedeme, provede se zavolání při každé výjimce (tedy i pokud má definovánu akci pro výjimku s konkrétním jménem). Funkci *testFunction()* se jako parametr předá název výjimky, zpráva, soubor a řádek na kterém byla výjimka vyvolána.

5.2.6 Ignorování chyb

V BugBusteru lze také nastavit které výjimky/chyby se mají ignorovat. Filtrování však

funguje jen u těch chyb, které nezpůsobí ukončení provádění skriptu. Filtrování lze využít především v případě, kdy je v aplikaci použita nějaká součást, která obsahuje velké množství chyb.

5.2.7 Zobrazení zdrojového kódu

Tato část je z celého výpisu nejčastěji používaná. Přehledným způsobem ukazuje, kde by mohla být chyba. Pro přehlednost bylo nezbytně nutné, aby BugBuster zvýrazňoval syntaxi a čísloval řádky. Jako nejjednodušší způsob vypadalo použití funkce *highlight_string()*. Požadoval jsem však některou další funkčnost, které šlo dosáhnout jen vlastním zvýrazňováním. Zvýrazňování funguje tak, že PHP kód rozloží na tokeny (pomocí funkce *token_get_all()*) a poté kód opět spojuje dohromady a označuje jej pomocí různých tříd. Takto označkovaný kód je pak nastylován pomocí CSS.

Použitím tohoto způsobu jsem tak mohl implementovat to, že když se myší najede například nad proměnnou, tato proměnná se zvýrazní zároveň na všech ostatních místech zdrojového kódu. Rozhodnutí, zda proměnnou zvýraznit či nikoliv je učiněno pouze na základě názvu proměnné. Nebere se tedy v úvahu výskyt stejné proměnné v jiné metodě.

Další funkcí, která je v základním nastavení vypnutá, je možnost přímé editace zdrojového kódu. Dvojklikem na daný řádek se změní na vstupní pole. Po úpravě kódu stačí klepnout na tlačítko "uložit" a provedené změny se uloží. Tato funkčnost je však velice nebezpečná v případě špatného nastavení. Pokud je ale webový server správně nastaven (nemá právo zapisovat do php skriptů), tak k žádnému narušení bezpečnosti nedojde. Pokud je nastavený produkční mód, tak je tato funkčnost automaticky vypnutá a nelze použít ani variantu s prefixem *prod_*.

5.3 Provoz na produkčním serveru

Při provozu na produkčním serveru je nežádoucí, aby chybové zprávy obsahovaly jakékoliv detailní informace o chybě. V BugBusteru jsou proto dvě možnosti jak chyby na serveru zobrazovat. Nejjednodušší způsob spočívá v tom, že se prostě jen zobrazí název chyby/výjimky.

Obr. 14 Výjimka na produkčním serveru

Druhý způsob je ještě méně detailní. Nezobrazí se již ani název chyby/výjimky. Místo toho se zobrazí pouze upravitelná chybová zpráva. Tento způsob používá většina velkých serverů (Google, Twitter atd.)

Obr. 15 Produkční chybová zpráva

V obou případech se detailní informace o chybě/výjimce zalogují do souboru. Poté je na nastavený email vývojáři zasláno upozornění, že došlo k nějaké chybě. Emailové upozornění se zašle pouze 1x (aby nedocházelo k zaplnění schránky). Po odeslání upozornění se vytvoří soubor *mail-sent.lock* jehož existence zabraňuje odeslání dalšího emailu. Vývojář si poté může na serveru detail chyby zobrazit a soubor ručně smazat.

5.4 Značkování v kódu (flags)

Při ladění aplikací (především kvůli výkonu) je třeba vědět, jak dlouho která část kódu trvá. Lze k tomu buď použít například rozšíření Xdebug, nebo v kódu na požadovaných místech zavolat metodu *Debug::flag()*. Prvním parametrem této metody je název značky. Druhým (volitelným) parametrem je skupina, do které značka patří.

Značkování funguje tak, že při inicializaci BugBusteru se spustí interní stopky. Od toho okamžiku se potom měří čas a spotřebovaná paměť.

Výsledky měření se poté zobrazí zavoláním metody *Debug::printFlags()*. Metoda má volitelný parametr, který určuje kterou skupinu značek vypsat.

```
Debug::flag('start');
sleep(1);
…
Debug::flag('test', 'my-type');
Debug::printFlags();
Debug::printFlags('my-type');
```

Př 15. Příklad použití značek při optimalizaci kódu

Flags ▼

#	Flag type	Flag name	Absolute time	Relative time	Absolute memory	Realtive memory
1	kDebug	Initializing kDebug class	0.000 ms	+0.000 ms	2.47MB	0B
2	default	start	1.834 ms	+1.834 ms	2.467MB	-3264B
3	default	after sleep(1)	1002.242 ms	+1000.408 ms	2.467MB	896B
4	default	after sleep(2)	3002.488 ms	+2000.246 ms	2.521MB	54.883kB
5	default	flag in function	3002.557 ms	+0.069 ms	2.522MB	904B
6	my-type	test	3002.613 ms	+0.056 ms	2.523MB	1.031kB
7	kDebug	before printing Flags	3002.741 ms	+0.128 ms	2.523MB	632B

Flags ▼

#	Flag type	Flag name	Absolute time	Relative time	Absolute memory	Realtive memory
6	my-type	test	3002.613 ms	+3002.613 ms	2.523MB	54.484kB

Obr. 16 Výpis použitých značek

5.5 Sledování proměnných

V PHP lze s rozšířením Xdebug sledovat proměnné podobně jako v dalších programovacích jazycích (C, C++,...). Je k tomu třeba vývojové prostředí, které práci s tímto rozšířením podporuje. Ne vždy však takovéto sledování funguje. Záleží vždy na koncepci projektu.

V rámci BugBusteru lze proměnné sledovat také. Sledování je však omezeno pouze na globální proměnné. Proměnné uvnitř objektů/funkcí sledovat nelze (nelze se žádným způsobem dostat do lokálního kontextu funkce).

PHP umožňuje zaregistrovat funkci, která se vykoná při každém kroku skriptu. Toho můžeme využít právě ke sledování proměnných. Toto sledování je však časově mnohem náročnější než při použití Xdebugu.

Samotné sledování funguje tak, že při prvním kroku se uloží hodnota sledované proměnné (může jich být samozřejmě i více) a v dalších krocích se aktuální hodnota proměnné (braná ze superglobálního pole $GLOBALS) porovnává s tou naposledy uloženou. Pokud se změnila, zjistí se název souboru a řádek, na kterém ke změně došlo a hodnota se uloží pro další porovnávání.

```
declare(ticks=1);
Debug::watch('testVariable');
$testVariable = 0;
$testVariable = 1;
…
Debug::showWatches();
```

Př 16. Použití sledování globálních proměnných

Velkou nevýhodou je, že v každém souboru, kde chceme proměnné sledovat, musí být uvedeno *declare(ticks=1);*. Bez uvedení této konstrukce by nefungovala právě funkce, která se stará o sledování proměnných. Zavoláním metody *showWatches()* si poté můžeme nechat zobrazit seznam sledovaných proměnných. Výpis obsahuje také název souboru a řádek, na kterém došlo ke změně.

Watches ▼

testVariable ▶

New value	File	Line
int(0)	C:\wamp\www\kDebug\example-watch.php	8
int(1)	C:\wamp\www\kDebug\example-watch.php	9
int(2)	C:\wamp\www\kDebug\example-watch.php	12
int(8)	C:\wamp\www\kDebug\example-watch.php	17
int(4)	C:\wamp\www\kDebug\example-watch.php	19

Obr. 17 Výpis sledované proměnné

Nemožnost sledovat proměnné uvnitř objektů a funkcí by se dala obejít například ručním voláním nějaké metody, které by se ovšem musela předávat název a hodnota proměnné. Toto řešení je však nepraktické, proto není v BugBusteru implementováno.

5.6 Logování

BugBuster sám automaticky zaznamenává chyby a výjimky do souboru. Zaznamenává je buď v rozšířené podobě (se všemi detaily), nebo ve zkrácené podobě (pouze název chyby, soubor a číslo řádku). Implementována je také podpora pro uživatelské logování zpráv.

```
Debug::log('test message');
```

Př 17. Zalogování zprávy do souboru

Zavolání metody *log()* způsobí zapsání zprávy do souboru *user.logs*. Adresář, kam se zprávy logují, lze nastavit v konfiguraci.

5.7 Stopky

Pro vnitřní potřebu měřit čas je v BugBusteru implementována rovněž podpora stopek. Prvním zavoláním metody *timer()* se stopky spustí (metoda vrátí *NULL*). Při dalším volání již metoda vrací rozdíl času od prvního volání metody. Jako volitelný parametr lze předat název. Lze tedy měřit několik událostí najednou. Výsledný čas je v sekundách.

```
Debug::timer('my-name');
sleep(1);
echo Debug::timer('my-name');
sleep(2);
echo Debug::timer('my-name');
```

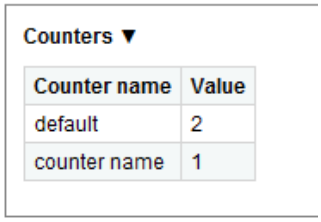
Př 18. Použití stopek

5.8 Počítadlo

Počítadlo je implementováno rovněž převážně pro vnitřní potřebu (část pro práci s FirePHP). Počítadlo funguje tak, že zavoláním metody *count()* se počítadlo se zadaným názvem (první parametr) zvýší o hodnotu zadanou v druhém parametru (standardně o 1). Hodnotu počítadla poté získáme metodou *getCounter()* nebo metodou *printCounters()*, která vypíše všechna počítadla v tabulce.

```
Debug::count('counter'); // 1
Debug::count('counter', 3); // 4
Debug::count('counter', -2); // 2
Debug::getCounter('counter');
Debug::printCounters();
```

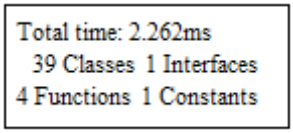
Př 19. Použití počítadla



Obr. 18 Zobrazení všech počítadel metodou printCounters()

5.9 Plovoucí informační panel

BugBuster rovněž dokáže zobrazovat plovoucí panel se základními informacemi o stránce. Zobrazí se zavoláním metody *Debug::printFloatingPanel();*. Volání by mělo být na samém konci stránky. Obsahuje informace o času zpracování skriptu, definovaných třídách, funkcích, konstantách a interfacech.



Obr. 19 Plovoucí informační panel

6 ZÁVĚR

BugBuster v žádném případě není dokonalý nástroj. Vždy se najde nějaká věc, která by šla vylepšit. Jako základní pomůcka pro vývoj aplikací se však velmi osvědčil. V praxi byl otestován na systému velkého e-shopu a pomohl odhalit několik skrytých problémů.

Nejzajímavější funkcí by byla možnost sledování proměnných. Bohužel se zde naráží na problém nemožnosti sledování proměnných uvnitř funkcí/metod. Tato skutečnost je při používání moderních metodik objektového programování zcela zásadní.

Další velice zajímavou funkcí je použití modifikátorů při výpisu proměnných. V žádném podobném projektu jsem tuto možnost prozatím neviděl. Tato funkčnost je již zcela připravena k reálnému použití. Výhodou je snadná rozšiřitelnost o další modifikátory.

Nejdůležitější část celé třídy, kvůli které jsem se do projektu vůbec pustil, je ta, která zachytává chyby a výjimky. K chybě se díky BugBusteru vývojář dozví další užitečné informace, které zejména po spuštění na produkčním serveru mohou být zásadní a urychlit odhalení chyby. Ve většině případů tak ušetří nemalé finanční prostředky ať už na platu vývojáře, tak na následných ztrátách v případě vzniku chyby. Vývojář je o chybě okamžitě informován emailem a může tak zahájit opravu chyby.

Při vývoji této třídy jsem prohloubil své znalosti v oblasti PHP aplikací a to především v oblasti práce s chybami. Rovněž bylo třeba se mnohem více soustředit na případné chyby v samotné ladící třídě. Třída, která obsluhuje zachytávání chyb, je sama nesmí obsahovat (jinak dojde v některých případech až k pádu webového serveru).

Ladící třída svými schopnostmi a rychlostí nemůže konkurovat rozšířením jako Xdebug. Hodí se však v případech kdy Xdebug nainstalovat nelze a především pro provoz na produkčních serverech kvůli své vylepšené práci s chybami.

SEZNAM POUŽITÉ LITERATURY

[1] GUTMANS, Andi; BAKKEN, Stig Saether; RETHANS, Derick. *Mistrovství v PHP5*. 1. vyd. Brno : CP Books, a.s., 2005. 656 s s. ISBN 80-251-0799-X. [kniha]

[2] LACKO, Luboslav. *PHP 5 a MySQL 5 : Hotová řešení*. 1. vyd. Brno : Computer Press, a.s., 2007. 320 s s. ISBN 80-251-1695-1. [kniha]

[3] ACHOUR, Mehdi, et al. *PHP* [online]. c1997, 2011-05-06 [cit. 2011-05-08]. PHP Manual. Dostupné z WWW: <http://www.php.net/manual/en/>. [webová stránka]

[4] FirePHP Wiki [online]. c2006, July 09, 2009 [cit. 2011-05-16]. FirePHP Protocol. Dostupné z WWW: <http://www.firephp.org/Wiki/Reference/Protocol>. [webová stránka]

[5] GRUDL, David. Nette Framework [online]. c2008, c2011 [cit. 2011-05-16]. NetteDebug. Dostupné z WWW: <http://doc.nette.org/cs/nette-debug>. [webová stránka]

[6] Xdebug [online]. c2002 [cit. 2011-05-19]. Xdebug: Documentation. Dostupné z WWW: <http://xdebug.org/docs/>. [webová stránka]