Bakalářská práce:

Kontinuální integrace při vývoji webových aplikací v PHP

# Úplně úvod

Cílem BP je připravit text, který umožní např. malému webovému studiu získat nezbytné teoretické znalosti pro implementaci CI platformy pro PHP a zároveň ho provede její praktickou implementací.

//Zkusit podle draftu ve Vergiliu, napsat o tom do BP

Obsah

[1 Úplně úvod 2](#_Toc317930090)

[2 Úvod 4](#_Toc317930091)

[3 Charakteristika konceptu Continuous Integration 5](#_Toc317930092)

[3.1 Techniky a zásady CI 5](#_Toc317930093)

[3.1.1 Jednotné úložiště zdrojových kódů 5](#_Toc317930094)

[3.1.2 Automatizované sestavení aplikace 6](#_Toc317930095)

[3.1.3 Testovatelný build 7](#_Toc317930096)

[3.1.4 Každý vývojář ukládá kód do úložiště alespoň jednou za den 7](#_Toc317930097)

[3.1.5 Po každém commitu proběhne sestavení aplikace na integračním stroji 7](#_Toc317930098)

[3.1.6 Build musí být rychlý 8](#_Toc317930099)

[3.1.7 Testování by se mělo provádět v prostředí co nejpodobnějším produkčnímu 8](#_Toc317930100)

[3.1.8 Každý má snadný přístup k informacím 8](#_Toc317930101)

[3.1.9 Automatizované nasazení 9](#_Toc317930102)

[3.2 Přínosy CI 9](#_Toc317930103)

[3.2.1 Snížení rizik 9](#_Toc317930104)

[3.2.2 Snížení množství manuálních činností 9](#_Toc317930105)

[3.3 Typický průběh práce vývojáře v prostředí CI 9](#_Toc317930106)

[3.4 Sestavení aplikace 9](#_Toc317930107)

[3.5 Testování 9](#_Toc317930108)

[3.6 Statická analýza kódu, kontrola kvality 10](#_Toc317930109)

[4 Výběr vhodného integračního serveru 11](#_Toc317930110)

[4.1 Přehled trhu s CI servery 11](#_Toc317930111)

[4.2 Jenkins / Hudson 11](#_Toc317930112)

[5 Srovnání CI nástrojů pro PHP 13](#_Toc317930113)

[5.1 PHP Lint 13](#_Toc317930114)

[5.1.1 Kontrola dopředné kompatibility 14](#_Toc317930115)

[5.1.2 Pre-commit hook 14](#_Toc317930116)

[5.2 PHP\_CodeSniffer 15](#_Toc317930117)

[5.3 PHP DCD 15](#_Toc317930118)

[5.4 PHP CPD 16](#_Toc317930119)

[5.5 PHP MD 17](#_Toc317930120)

[5.6 PHP Depend 17](#_Toc317930121)

[5.7 Neinicializované proměnné 17](#_Toc317930122)

[5.8 PHPLOC 17](#_Toc317930123)

[5.9 Docblox 18](#_Toc317930124)

[5.10 JS lint? 18](#_Toc317930125)

[6 Praktická implementace CI platformy v malé firmě 19](#_Toc317930126)

[7 Závěr 21](#_Toc317930127)

[8 Seznamy 22](#_Toc317930128)

[9 Seznam použité literatury a zdrojů 23](#_Toc317930129)

# Úvod

Při vývoji softwarového projektu většinou jednotliví vývojáři nebo jednotlivé vývojářské týmy pracují na samostatných subsystémech, které je poté nutné propojit dohromady. Tento proces se označuje jako integrace. Tradiční metodiky vývoje software jako je například Vodopádový model (@todo zdroj) zařazují fáze integrace, testování a kontroly kvality až po dokončení fáze vývoje (viz Obrázek 1.1). Doba samotného vývoje software je často odhadnuta nesprávně, i když tomu lze částečně předejít. [Odhadování SW projektů, @todo citace].

Obrázek . Schéma vodopádového modelu vývoje software

//co dát obrázek z nějaké knížky? Ať mám další citaci

Odhad doby trvání integrační a QA[[1]](#footnote-1) fáze je vzhledem k jejich neurčitosti prakticky nemožný. Fáze s neznámou dobou trvání v závěru projektu, kdy už ostatní fáze mohly překročit odhadované doby trvání, může způsobit překročení plánované doby trvání celého projektu a tedy i rozpočtu.

S řešením problému neurčitosti integrační fáze přichází technika tzv. kontinuální (neboli průběžné) integrace. Fáze integrace, testování a kontroly kvality se rozloží do celého průběhu fáze implementace a jsou prováděny průběžně, vždy po každé změně programového kódu.

# Charakteristika konceptu Continuous Integration

Kontinuální integrace zavádí proces průběžné kontroly kvality kódu – často prováděné nenáročné úkony, místo kontroly kvality až po dokončení vývoje.[[2]](#footnote-2)

[Fowler, 2006] definici dále upřesňuje:

Continous Integration[[3]](#footnote-3) je technika používaná při vývoji software, kdy členové týmu integrují svůj kód často, obvykle aspoň jednou za den, což vede k mnoha integracím každý den. Každá integrace je ověřená automatickým sestavením (včetně testů), aby byly případné problémy objeveny co nejdříve. Mnoho týmů zjistilo, že tento přístup vede k výraznému snížení problémů při integraci a umožňuje jim rychleji dodávat kvalitnější software.[[4]](#footnote-4)

Kontinuální integrace byla poprvé popsána[[5]](#footnote-5) jako jedna z praktik tzv. Extrémního programování[[6]](#footnote-6), nicméně je ji samozřejmě možné využít i bez implementace ostatních praktik EP.

## Techniky a zásady CI

Kontinuální integrace je souhrnem několika technik, které lze do týmového workflow zahrnout samostatně (a tedy postupně). V této kapitole a jejích podkapitolách shrnu jednotlivé techniky podle [Fowler, 2006] a [Duvall, et al., 2007].

### Jednotné úložiště zdrojových kódů

Každý softwarový projekt se skládá z mnoha souborů různých typů – soubory se zdrojovým kódem, konfigurační soubory, multimediální data (například obrázky, které jsou součástí webu). Při spolupráci více lidí na projektu může být problém zajistit, aby všichni měli všechny soubory v aktuální verzi. Dříve bylo běžné na sdílení projektových souborů používat sdílené síťové disky, ale toto řešení neumožňuje snadno získat starší verzi souboru se zdrojovým kódem.

Tento problém řeší nástroje pro správu zdrojového kódu[[7]](#footnote-7), které kromě úložiště (Repository) nabízejí i přístup ke starším verzím souborů. Úložiště může být buď centrální (VCS) anebo distribuované (DVCS). Mezi nejznámější open-source nástroje pracující s centrálním úložištěm patří Subversion[[8]](#footnote-8) nebo dnes již zastaralé CVS[[9]](#footnote-9). Z komerčních lze jmenovat Rational Team Concert[[10]](#footnote-10) od IBM nebo Team Foundation Server[[11]](#footnote-11) od společnosti Microsoft. Mezi nejznámější open-source nástroje pracující s distribuovaným úložištěm patří Git[[12]](#footnote-12) (je používaný pro verzování jádra operačního systému Linux), Mercurial[[13]](#footnote-13) nebo Bazaar[[14]](#footnote-14). Z komerčních lze uvést BitKeeper[[15]](#footnote-15) od společnosti BitMover, Inc. (používaný pro verzování jádra operačního systému Linux v letech 2002-2005[[16]](#footnote-16)).

Jakmile tým používá jednotné úložiště zdrojového kódu, odpadají problémy s dohledáváním chybějících souborů. To předpokládá, že do repositáře se ukládá opravdu vše – i konfigurační soubory, databázová schémata, instalační skripty a knihovny třetích stran. Fowler doporučuje jednoduché pravidlo: Pokud na čistě nainstalovaném počítači provedete checkout[[17]](#footnote-17) repositáře, tak by mělo být možné provést sestavení aplikace[[18]](#footnote-18). Výjimku tvoří například operační systém, JDK[[19]](#footnote-19) nebo databázový systém. Fowler také doporučuje do úložiště ukládat i další soubory, které nejsou nezbytně nutné k sestavení aplikace, ale mohou ušetřit práci ostatním členům týmu – například konfigurační soubory pro IDE[[20]](#footnote-20).

Nicméně například [Menšík, 2009] s Fowlerem nesouhlasí a považuje ukládání knihoven do verzovacího systému za zbytečné. Zároveň ale zmiňuje nutnost umožnit skriptu pro sestavení přístup ke sdílenému síťovému disku s knihovnami.

V úložišti by mělo být uloženo vše, co je potřeba pro sestavení aplikace, ale nic z toho, co vznikne při sestavení – podle Fowlera to většinou znamená, že není snadné provést spolehlivé sestavení aplikace.

### Automatizované sestavení aplikace

Převod zdrojových kódů do funkční aplikace je často složitý proces, který zahrnuje kompilaci zdrojových kódů, přesouvání souborů, nahrávání schémat do databáze a další. Vzhledem k tomu, že tento proces lze automatizovat, tak by také automatizovaný měl být. Pokud ho provádí člověk ručně, tak zbytečně ztrácí čas a může se stát, že udělá chybu.

Mezi systémy pro automatizované sestavení aplikace lze zařadit make[[21]](#footnote-21) používaný v operačních systémech Linux a Unix, Ant[[22]](#footnote-22) používaný pro projekty psané v jazyce Java nebo Phing[[23]](#footnote-23) používaný pro projekty psané v jazyce PHP.

Častou chybou je, že automatizovaný build nezahrnuje všechny nutné kroky a je nutné manuálně provést některé další úkony po skončení automatizovaného sestavení. Fowler zde doplňuje svoje pravidlo: Pokud na čistě nainstalovaném počítači provedete checkout repositáře, tak by mělo být možné pomocí jednoho příkazu získat běžící aplikaci.

IDE často obsahují nástroje na provádění sestavení, ale jejich použití je problematické, protože soubory s popisem procesu sestavení jsou většinou proprietární pro dané IDE a není je možné bez něj spustit. Samozřejmě je možné, aby jednotliví vývojáři spouštěli lokální buildy přímo z IDE, ale vždy potřeba mít skript pro sestavení, který lze spustit na serveru a je nezávislý na IDE.

#### Upřesnění pro projekty vyvíjené v jazyce PHP

Skripty psané v PHP se většinou nekompilují předem, ale až při zpracování v interpreteru. Proto skript pro sestavení aplikace nebude, na rozdíl například od projektů psaných v jazyce Java, obsahovat kompilaci, ale jen kontrolu syntaktické správnosti zdrojových kódů.

### Testovatelný build

Úspěšné sestavení neznamená, že aplikace bude fungovat správně. Pro odchycení chyb je vhodné do procesu sestavení zařadit i automatizované testování. Vhodným, nicméně ne nezbytným, přístupem je proto Test Driven Development[[24]](#footnote-24), kdy se nejdříve píše test a teprve poté kód aplikace. TDD je jednou z klíčových praktik EP [Beck, 2005].

Nejčastěji používanými testovacími nástroji jsou tzv. xUnit[[25]](#footnote-25) frameworky, kde x značí programovací jazyk. Existuje tedy JUnit[[26]](#footnote-26) pro Javu, NUnit[[27]](#footnote-27) pro .NET, PHPUnit[[28]](#footnote-28) pro PHP a další.

Testy jsou většinou sdružovány do sad testů a po jejich spuštění je vygenerován report, který informuje o tom, zda některé testy selhaly.

Fowler [Fowler, 2006] upozorňuje na to, že testy neobjeví všechny chyby. Nicméně nedokonalé testy, které jsou spouštěny často, jsou podle něj mnohem lepší než dokonalé testy, které nejsou nikdy napsány.

### Každý vývojář ukládá kód do úložiště alespoň jednou za den

Nejsnazším způsobem, jak může dát vývojář vědět ostatním kolegům v týmu o změnách v nějaké části aplikace je uložení změn do úložiště. Do úložiště je samozřejmě vhodné ukládat jen funkční kód, což nutí vývojáře rozdělit velký úkol na mnoho dílčích podúkolů, které lze řešit samostatně. To že je kód funkční ověří vývojář sestavením aplikace a spuštěním automatizovaných testů. Těsně před uložením změn si musí vývojář stáhnout případné změny, které provedli ostatní členové týmu a testy ověřit, že jeho kód stále funguje i s těmito změnami. Teprve poté může kód uložit.

Pokud by při stažení nejnovějších změn z úložiště narazil na nějaký konflikt, tak většinou nebude problém ho vyřešit, protože během těch několika hodin od minulého stažení nemohlo změn být mnoho.

### Po každém commitu proběhne sestavení aplikace na integračním stroji

Některé společnosti provádějí sestavení aplikace v naplánovaných intervalech (např. každou noc), ale to není úplně ideální, protože cílem kontinuální integrace je odhalit chyby co nejdříve je to možné. Pokud sestavení probíhá jednou za 24 hodin, může se stát, že chyby v systému zůstanou dlouhou dobu (a budou působit problémy ostatním členům týmu). Proto je mnohem lepší provádět sestavení po každé změně v úložišti. To může probíhat buď manuálně, nebo automaticky.

Při manuálním sestavení si vývojář po uložení změn do úložiště sedne k speciálnímu počítači, který je vyhrazen jen na integraci[[29]](#footnote-29). Stáhne si aktuální verzi z úložiště, spustí sestavení, a pokud to proběhne správně, tak je jeho úkol hotový.

Druhou variantou je využití integračního serveru, který sleduje úložiště (v pravidelných intervalech se dotazuje úložiště – tzv. polling), a po každé zjištěné změně spustí build a pošle autorovi e-mail s výsledkem.

Integrační server není nezbytně nutný, ale jeho použití je pohodlnější. Jeho další výhodou je, že pokud součástí buildu je i statická analýza zdrojového kódu[[30]](#footnote-30), můžeme na výstupu CI serveru sledovat dlouhodobé trendy v kvalitě kódu.

### Build musí být rychlý

Cílem kontinuální integrace je rychle poskytnout vývojářům zpětnou vazbu. Proto by sestavení na integračním serveru mělo být co nejrychlejší. Kent Beck [Beck, 2005] ve své metodice extrémního programování doporučuje, aby build netrval déle než 10 minut. Pokud by trval déle, výrazně se zvyšuje šance, že ho vývojáři nebudou spouštět jako lokální build.

Často je sestavení zpomalováno testy, které pracují s externími zdroji, jako je databáze nebo služby. V takovém případě je možné build rozdělit na dvě části – primární (spouští se po commitu do úložiště), sekundární (spouští se po úspěšném doběhnutí primárního). Výhoda tohoto přístupu je v tom, že vývojáři stačí, když počká na výsledek primárního sestavení. Pokud by sekundární build zjistil nějaké problémy, tak to není tak velký problém, protože testy důležité funkčnosti jsou zahrnuté v primárním sestavení.

Pokud by sekundární build trval moc dlouho, tak je možné využití například více serverů, na kterých testy poběží (integrační servery často funkčnost pro distribuované sestavení a testování obsahují). V tu chvíli musí být testy schopny běžet izolovaně. Hlubší rozbor možností paralelního spouštění testů na více strojích nicméně přesahuje rozsah této práce.

### Testování by se mělo provádět v prostředí co nejpodobnějším produkčnímu

Cílem testování je odhalit problémy, které by mohly nastat v produkčním prostředí. Proto je vhodné, aby testovací prostředí bylo co nejvíce podobné produkčnímu. Pokud se budou lišit, tak se může stát, že se některé chyby nepodaří odhalit nebo naopak bude docházet k planým poplachům.

Například pokud by vývojáři pracovali na počítačích s operačním systémem Microsoft Windows, integrační server by také používal Windows, ale na produkčním serveru by aplikace běžela na operačním systému Linux, neprojevily by se problémy s rozdílnými znaky pro konce řádků nebo oddělovače adresářů v cestě.

Proto by integrační server měl běžet na stejném operačním systému, používat stejné verze aplikací, běhových prostředí i knihoven.

### Každý má snadný přístup k informacím

Při aplikaci kontinuální integrace je důležitá komunikace mezi členy týmu. Jednou z nejdůležitějších informací je stav posledního sestavení, protože pokud to bylo neúspěšné (což značí, že kód obsahuje chybu nebo chyby), tak není vhodné, aby si vývojář aktualizoval svoji kopii zdrojových kódů z úložiště na tuto verzi.

Pokud se využívá manuální integrace na integračním stroji, tak by na jeho monitoru mělo vždy být vidět stav posledního sestavení.

Automatické nástroje pro kontinuální integraci umožňují snadnější přístup k této informaci. Členové geograficky distribuovaných vývojových týmů mohou dostávat tuto informaci pomocí e-mailu, SMS, aplikace běžící na počítači vývojáře nebo pomocí pluginu do IDE. Týmy často experimentují s vizuálními pomůckami, které ukazují aktuální stav buildu. Některé používají například červenou a zelenou lávovou lampu[[31]](#footnote-31), LED diody řízení Arduinem[[32]](#footnote-32) nebo detailnější tabuli s přehledem stavu buildů[[33]](#footnote-33)

### Automatizované nasazení

Využívání kontinuální integrace umožňuje častěji získávat stabilní sestavení aplikace, čehož lze využít pro zvýšení frekvence vydávání a nasazování nových verzí. Proto je vhodné mít nasazování nových verzí řešeno také pomocí automatizovaného skriptu. Technika častého nasazování nových verzí se označuje jako Continuous Deployment nebo také jako Continuous Delivery.

Podrobný rozbor tématu však přesahuje rozsah této práce. Více informací lze nalézt například v [Humble, a další, 2010], případně se zaměřením jen na PHP v [Zikmund, 2011]. //@todo přečíst jestli je OK ☺

## Přínosy CI

Využití kontinuální integrace může týmu i celému procesu vývoje daného software přinést různá zlepšení. V této kapitole shrnu možné přínosy využití CI podle toho, jak je rozebral [Duvall, et al., 2007 pp. 29-32].

### Snížení rizik

### Snížení množství manuálních činností

@todo

## Typický průběh práce vývojáře v prostředí CI

1. Vývojář si z úložiště stáhne nejnovější verzi zdrojových kódů
2. Provede potřebné úpravy - většinou trvají jen několik hodin – všechny úkoly jsou rozložené na dílčí podúkoly
3. Spustí soukromé sestavení aplikace (včetně testů) a ověří, že vše funguje
4. Stáhne si z úložiště nejnovější změny zdrojových kódů od ostatních členů týmus
5. Pomocí soukromého sestavení aplikace ověří, že vše stále funguje
6. Uloží své změny do úložiště
7. Integrační server provede sestavení aplikace a spuštění testů a zašle informační e-mail
8. Pokud nedojde k žádným chybám, vývojář si úkol může označit jako dokončený

## Sestavení aplikace

## Testování

## Statická analýza kódu, kontrola kvality

Statickou analýzou zdrojového kódu označujeme proces získávání metrik kvality kódu. V kontextu kontinuální integrace získáváme vybrané metriky automatizovaně a při využití vhodných nástrojů můžeme sledovat jejich dlouhodobé trendy. [co se nedá měřit, to se nedá řídit @todo] Sledování a vyhodnocování metrik zdrojového kódu je důležité, protože nám může pomoci odhalit tzv. code smell (zatuchlý kód), který může způsobovat komplikace v pozdějších fázích softwarového projektu.

Mezi nejčastěji získávané metriky patří

* Dodržování Coding Standards (standardů pro psaní kódu)
* Množství duplicitní kód

//Lepší kód -> větší zábava při programování -> spokojenější programátoři -> menší fluktuace

# Výběr vhodného integračního serveru

Jak již jsem zmiňoval dříve, kontinuální integraci je možné provádět i bez využití integračního serveru, nicméně nenapadá mě žádný důvod proč. Většina serverů pro kontinuální integraci je k dispozici zdarma a jejich využití přináší další automatizaci úkonů, které by bylo jinak nutné dělat ručně. Zároveň máme jistotu, že je sestavení spouštěno vždy stejně.

## Přehled trhu serverů pro kontinuální integraci

Na trhu jsou k dispozici různé CI servery. Ten nejstarší, CruiseControl[[34]](#footnote-34), byl vyvinut ve společnosti ThoughtWorks, pro kterou pracuje Martin Fowler, autor známého článku [Fowler, 2006]. Existuje rozšíření phpUnderControl[[35]](#footnote-35), které usnadňuje prvotní nastavení CI nástrojů pro PHP, nicméně již není aktivně vyvíjeno, takže jeho nasazení bych nedoporučoval. Kromě toho existují další méně známé CI servery, například Xinc[[36]](#footnote-36), který je napsaný v jazyce PHP.

@todo dopsat další <http://en.wikipedia.org/wiki/Comparison_of_Continuous_Integration_Software>

Zároveň bych sem zařadil i Sismo[[37]](#footnote-37), což je Continuous Testing Server – zaměřuje se jen na spouštění testů. Díky tomu ho lze využít jako komplement k CI serveru pro rychlé spouštění jednotkových testů lokálně, po commitu do repositáře ve verzovacím systému GIT. Podobně funguje i Travis CI[[38]](#footnote-38), jehož cílem je vytvořit distribuovaný systém, pro spouštění testů open-source software.

Z komerčních CI serverů bych zde uvedl Bamboo[[39]](#footnote-39) od společnosti Atlassian[[40]](#footnote-40). Výhodou tohoto CI serveru je snadná integrace s dalšími nástroji pro projektové řízení od této společnosti.

## Jenkins / Hudson

Velmi známým integračním serverem je Jenkins (resp. Hudson). Někdy dochází k zaměňování těchto názvů, proto by bylo vhodné to zde upřesnit. Jeden ze zaměstnanců Sun Microsystems v roce 2006 vyvinul integrační server Hudson a uvolnil ho pod svobodnou licencí MIT. V následujících letech se vývoje účastnili i další vývojáři. V lednu 2010 společnost Oracle koupila[[41]](#footnote-41) společnost Sun Microsystems, čímž mj. získala název Hudson, který si později registrovala jako ochrannou známku. Oracle se během sjednocování infrastruktury s ostatními open-source projekty, které zastřešují, dostal do sporu[[42]](#footnote-42) s komunitou vývojářů Hudsonu a ti se rozhodli zbavit závislosti na společnosti Oracle[[43]](#footnote-43) a začali Hudson dále vyvíjet pod názvem Jenkins[[44]](#footnote-44).

Společnost Oracle dál vyvíjí Hudson, ale komunita se spolu s hlavními vývojáři přesunula k Jenkinsu, takže rozvoj Hudsonu se zpomalil. Oba projekty jsou v současné době hostovány na Githubu, takže je možné snadno provést srovnání rychlosti vývoje a zájmu vývojářů o ně:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Projekt** | **URL** | **Sledování** | **Forky** | **Pull Requesty** | **Commity v roce 2012** |
| Hudson | <https://github.com/hudson/hudson/> | 159 | 34 | 0 | 6 |
| Jenkins | <https://github.com/jenkinsci/jenkins/> | 1112 | 398 | 16 | 257 |

(hodnoty jsou k 25. 2. 2012)

Z těchto hodnot je vidět, že Jenkins je aktivně vyvíjen a zároveň se těší velkému zájmu uživatelů, což ho dělá vhodnější volbou než Hudson. Při výuce vývoje v Javě na VŠE se dříve využíval integrační server Hudson, nicméně dnes (25. 2. 2012) již to je Jenkins[[45]](#footnote-45), což potvrzuje moji domněnku, že Jenkins je vhodnější volbou než Hudson.

Vzhledem k tomu, že většina zdrojů, @todo, které se kontinuální integrací v prostředí PHP zabývají, zvolila CI server Jenkins, nenašel jsem důvod, proč vybrat jiný.

# Srovnání CI nástrojů pro PHP

CI nástroje použitelné pro jazyk PHP lze rozdělit do tří velkých skupin:

* nástroje na kontrolu kvality zdrojových kódů pomocí statické analýzy
* nástroje na automatizované testování
* podpůrné nástroje pro automatizaci, tvorbu dokumentace, schématu tříd

## PHP Lint

Kontrolu kvality kódu nemá smysl provádět dříve, než si jsme jisti, že napsaný kód je syntakticky v pořádku.

@todo Nástroje pro kontrolu kvality by s tím mohly mít problém - zkusit

Některá moderní IDE jako Zend Studio[[46]](#footnote-46), Eclipse PDT[[47]](#footnote-47) nebo Netbeans[[48]](#footnote-48) už sice kontrolu syntaktických chyb obsahují (viz Obrázek 4.1), ale slouží spíše jako upozornění pro vývojáře. Integrační nástroj nemůže spoléhat na to, že si vývojář chyby všimne a opraví ji.



Obrázek . Kontrola syntaktických chyb v editoru Zend Studio 8

Samotné PHP umožňuje provést kontrolu syntaxe PHP souboru pomocí přepínače '-l' (lint):

> php –help

-l Syntax check only (lint)

Pokud kontrolu spustíme na skriptu test1-error.php, který obsahuje syntaktickou chybu, dostaneme chybový výstup (viz Kód 4.2)

<?php

echo "Kontinuální integrace" //chybí středník

Kód . test1-error.php

> php -l test1-error.php

Parse error: syntax error, unexpected $end, expecting ',' or ';' in test1-error.php on line 2

Errors parsing test1-error.php

Kód . Chybový výstup z PHP Lint

Pokud je soubor syntakticky v pořádku (viz Kód 4.3), vrátí PHP Lint výstup (Kód 4.4)

<?php

echo "Kontinuální integrace";

Kód . Zdrojový kód souboru test2-ok.php

> php -l test2-ok.php

No syntax errors detected in test2-ok.php

Kód . Výstup z PHP Lint, když je kód v pořádku

### Kontrola dopředné kompatibility

PHP Lint můžeme použít také pro kontrolu, zda zdrojové kódy budou zkompilovatelné i v další verzi PHP. Problém může nastat například, pokud jako identifikátor použijeme slovo, které je v další verzi zařazeno mezi klíčová slova.

Pokud provedeme kontrolu pomocí PHP Lint (verze PHP 5.3) na souboru (viz Kód 4.5), tak je vše v pořádku (Kód 4.6).

<?php

class Trait {}

Kód . test3-trait.php

>php -l test3-trait.php

No syntax errors detected in test3-trait.php

Kód . Výstup z PHP Lint (verze PHP 5.3) na souboru test3-trait.php

Pokud ovšem použijeme PHP ve verzi 5.4.0alpha2, kde trait je klíčovým slovem, dostaneme chybový výstup (viz Kód 4.7)

> php -l test3-trait.php

PHP Parse error: syntax error, unexpected 'Trait' (T\_TRAIT), expecting identifier (T\_STRING) in test3-trait.php on line 2

Parse error: syntax error, unexpected 'Trait' (T\_TRAIT), expecting identifier (T\_STRING) in test3-trait.php on line 2

Errors parsing test3-trait.php

Kód . Chybový výstup z PHP Lint (verze PHP 5.4.0alpha2) na souboru test3-trait.php

Pro účely CI je důležité spouštět kontrolu syntaktické správnosti pro aktuální verzi PHP, ale může být užitečné spouštět i kontrolu, zda skripty bude možné spustit v novější verzi PHP. Vzhledem k tomu, že novější verze PHP jsou většinou rychlejší[[49]](#footnote-49), můžeme díky zajištění dopředné kompatibility snadno získat výkonové zlepšení. Podle [Bergmann, 2008] je PHP 5.3 1,2x rychlejší než PHP 5.2, takže pokud by projekt vyvíjený v době PHP 5.2 měl podobně kontrolovanou kompatibilitu s PHP 5.3, mohl jen pomocí aktualizace na novou verzi PHP získat 20% zrychlení běhu.

### Pre-commit hook

Systémy pro správu verzí jako Subversion nebo GIT umožňují navázat spouštění skriptů na události vyvolané v různých fázích ukládání souborů do úložiště. Pro kontrolu syntaktické správnosti souborů ještě před jejich přijetím do úložiště využijeme pre-commit hook, který může probíhající commit zrušit. Lze využít už hotové skripty pro SVN[[50]](#footnote-50) nebo GIT[[51]](#footnote-51). Oba fungují tak, že získají seznam změněných souborů s příponou PHP, spustí na nich kontrolu syntaktické správnosti a pokud jsou všechny soubory v pořádku, tak commit umožní dokončit. V opačném případě vrátí textovou informaci, které soubory obsahují syntaktické chyby.

PHP kód nemusí být ovšem uložený jen v souborech s příponou PHP. Například často používaný PHP framework Zend Framework[[52]](#footnote-52) využívá jako šablonovací jazyk přímo PHP a šablony jsou ve výchozím nastavení uloženy v souborech s příponou PHTML (viz Kód 4.8). Proto by bylo vhodné, aby se během commitu kontrolovala i jejich syntaktická správnost, což výše uvedené skripty neřeší.

<h1><?php echo "Kontinuální integrace"; ?></h1>

Kód . Příklad šablony v PHTML souboru (test4-template.phtml)

## PHP\_CodeSniffer

http://pear.php.net/package/PHP\_CodeSniffer

Coding standards neboli Standardy pro psaní kódu označuje soubor pravidel jak formátovat zdrojový soubor..

precommit hook na coding standards - pomalé

## PHP DCD

Dead Code Detector (DCD) for PHP code[[53]](#footnote-53) je nástroj, který v PHP souborech odhalí již nepoužívané části kódu (tzv. mrtvý kód). Není tak problém zjistit, které funkce (viz Kód 4.9 a Kód 4.10) nebo metody (viz Kód 4.11 a Kód 4.12) již nejsou odnikud volané.

<?php

function notCalled() {

return false;

}

function called() {

return true;

}

var\_dump(called());

Kód . phpdcd01-global-functions.php

> phpdcd phpdcd01-global-functions.php

phpdcd 0.9.2 by Sebastian Bergmann.

- notCalled()

declared in phpdcd01-global-functions.php:2

Kód . Výstup při kontrole „mrtvého kódu“ v souboru phpdcd01-global-functions.php

<?php

class Test {

function notCalled() {

return true;

}

function called() {

return true;

}

}

$t = new Test();

$t->called();

Kód . phpdcd02-class.php

> phpdcd phpdcd02-class.php

phpdcd 0.9.2 by Sebastian Bergmann.

- Test::notCalled()

declared in phpdcd02-class.php:3

Kód . Výstup při kontrole „mrtvého kódu“ v souboru phpdcd02-class.php

Nicméně nástroj má zatím problém s privátními nebo protected metodami, které hlásí jako nepoužívané, i když jsou v kódu volané (viz Kód 4.13 a Kód 4.14). Chyba již autorovi byla nahlášena[[54]](#footnote-54), tak ji snad brzy opraví. Dalším problémem je to, že phpdcd provádí jen statickou analýzu kódu a nerozpozná některá dynamická volání (Reflection API, call\_user\_func, použití proměnné pro název třídy a další[[55]](#footnote-55)).

<?php

class Test {

private function isOK() {

return false;

}

public function printStatus() {

var\_dump($this->isOK());

}

}

$a = new Test();

$a->printStatus();

Kód . phpdcd03-bug-private-method.php

> phpdcd phpdcd03-bug-private-method.php

phpdcd 0.9.2 by Sebastian Bergmann.

- Test::isOK()

declared in phpdcd03-bug-private-method.php:3

Kód . chybný výstup při kontrole „mrtvého kódu“ v souboru phpdcd03-bug-private-method.php

Přesnost nástroje tedy sice není stoprocentní, ale určitě má smysl ho použít, i kdyby odhalil jen některé, již nepoužívané části kódu.

Bohužel nástroj zatím nemá možnost výstupu do XML použitelného pro CI server, nicméně už to bylo zadáno jako issue[[56]](#footnote-56) a zkusím to vyřešit v rámci praktické části práce. @todo (možná)

## PHP CPD

PHP Copy/Paste Detector[[57]](#footnote-57) je nástroj, který odhalí zkopírované bloky kódu. Jeho použití je jednoduché (viz Kód 4.15).

> phpcpd .

phpcpd 1.3.2 by Sebastian Bergmann.

Found 1 exact clones with 24 duplicated lines in 1 files:

- ArticlesController.php:14-38

ArticlesController.php:67-91

20.34% duplicated lines out of 118 total lines of code.

Kód . Ukázka použití PHP CPD

Samozřejmostí je výstup do XML použitelného pro CI server, který lze aktivovat přepínačem --log-pmd <file>.

## PHP MD

<http://phpmd.org/rules/index.html>

## PHP Depend

<http://pdepend.org/>

## Neinicializované proměnné

http://code.google.com/p/php-initialized/

## PHPLOC

PHPLOC[[58]](#footnote-58) je jednoduchý nástroj, který umožňuje získat přehled o počtu řádků kódu v projektu a počítá některé další metriky (@todo rozpracovat). Jeho použití je jednoduché (viz Kód 4.16).

> phploc ./application

phploc 1.6.1 by Sebastian Bergmann.

Directories: 59

Files: 290

Lines of Code (LOC): 44338

Cyclomatic Complexity / Lines of Code: 0.05

Comment Lines of Code (CLOC): 8176

Non-Comment Lines of Code (NCLOC): 36162

Namespaces: 0

Interfaces: 0

Classes: 290

Abstract: 6 (2.07%)

Concrete: 284 (97.93%)

Average Class Length (NCLOC): 129

Methods: 1182

Scope:

Non-Static: 1164 (98.48%)

Static: 18 (1.52%)

Visibility:

Public: 796 (67.34%)

Non-Public: 386 (32.66%)

Average Method Length (NCLOC): 31

Cyclomatic Complexity / Number of Methods: 2.48

Anonymous Functions: 0

Functions: 10

Constants: 280

Global constants: 2

Class constants: 278

Kód . Ukázkový výstup z aplikace phploc

Samozřejmostí je opět výstup do XML použitelného pro CI server, který lze aktivovat přepínačem ‑‑log-xml <file>.

## Docblox

* Kontrola docbloků+ schéma

http://www.docblox-project.org/

## JS lint?

Možná je nějaký modul v PHP CS

http://zdrojak.root.cz/clanky/kontrola-javascriptu-s-jslint-a-jshint/

http://www.tomas-dvorak.cz/clanky/jshint-a-spousteni-validace-javascriptu-z-prikazove-radky

# Praktická implementace CI platformy v malé firmě

Jako operační systém pro instalaci CI platformy jsem zvolil Linux, konkrétně distribuci Debian (http://www.debian.org/). Instalace samotného operačního systému překračuje rozsah této práce, proto v dalších krocích budu předpokládat nainstalovaný základní verzi operačního systému Debian ve verzi 6.

hostname: jenkins

root: martin

martin: root

apt-get update

apt-get upgrade

apt-get install htop

apt-get install php-pear

Automatizace buildu

----------------------

Jak již jsem popisoval v teoretické části, kontinuální integrace vyžaduje automatizovaný build. (@todo Jenkins). Integrační server Jenkins, který jsem zvolil pro implementaci podporuje

* CI server řeší automatické spouštění buildu 🡪 nejdříve potřebuji build skript v Phingu, který se bude dát spouštět lokální (multiplatformní - Win / Lin)
* nastavení od A-Z
* Úprava procesů
* praktické použití na nějakém projektu
* //integrační problémy při použití branchí

TODO

* Seznam zkratek

# Závěr

* Má Zend Framework Continous Integration server?
* <http://www.youbrokethebuild.com/>
* Continuous Integration anti-patterns <http://www.ibm.com/developerworks/java/library/j-ap11297/>

# Seznamy

[Obrázek 1.1 Schéma vodopádového modelu vývoje software 2](file:///I:\_BP\BP.docx#_Toc299543608)

[Obrázek 4.1 Kontrola syntaktických chyb v editoru Zend Studio 8 9](#_Toc299543609)

[Kód 4.1 test1-error.php 9](#_Toc299623877)

[Kód 4.2 Chybový výstup z PHP Lint 9](#_Toc299623878)

[Kód 4.3 Zdrojový kód souboru test2-ok.php 10](#_Toc299623879)

[Kód 4.4 Výstup z PHP Lint, když je kód v pořádku 10](#_Toc299623880)

[Kód 4.5 test3-trait.php 10](#_Toc299623881)

[Kód 4.6 Výstup z PHP Lint (verze PHP 5.3) na souboru test3-trait.php 10](#_Toc299623882)

[Kód 4.7 Chybový výstup z PHP Lint (verze PHP 5.4.0alpha2) na souboru test3-trait.php 10](#_Toc299623883)

[Kód 4.8 Příklad šablony v PHTML souboru (test4-template.phtml) 11](#_Toc299623884)

[Kód 4.9 phpdcd01-global-functions.php 11](#_Toc299623885)

[Kód 4.10 Výstup při kontrole „mrtvého kódu“ v souboru phpdcd01-global-functions.php 11](#_Toc299623886)

[Kód 4.11 phpdcd02-class.php 12](#_Toc299623887)

[Kód 4.12 Výstup při kontrole „mrtvého kódu“ v souboru phpdcd02-class.php 12](#_Toc299623888)

[Kód 4.13 phpdcd03-bug-private-method.php 12](#_Toc299623889)

[Kód 4.14 chybný výstup při kontrole „mrtvého kódu“ v souboru phpdcd03-bug-private-method.php 12](#_Toc299623890)

# Seznam použité literatury a zdrojů

**Beck, Kent. 2005.** *Extreme Programming Explained: Embrace Change.* Boston, MA : Addison-Wesley, 2005. ISBN: 0321278658.

**—. 2002.** *Test-driven Development: By Example.* Boston : Addison-Wesley Professional, 2002. ISBN: 0321146530.

**Bergmann, Sebastian. 2008.** Benchmark of PHP Branches 3.0 through 5.3-CVS. [Online] 7. Únor 2008. [Citace: 20. Červenec 2011.] http://sebastian-bergmann.de/archives/745-Benchmark-of-PHP-Branches-3.0-through-5.3-CVS.html.

**Duvall, Paul, Matyas, Stephen and Glover, Andrew. 2007.** *Continuous Integration: Improving Software Quality and Reducing Risk.* Upper Saddle River, NJ : Addison-Wesley Professional, 2007. http://www.amazon.com/Continuous-Integration-Improving-Software-Reducing/dp/0321336380. ISBN: 978-0321336385.

**Fowler, Martin. 2006.** Continuous Integration. [Online] 1. Květen 2006. [Citace: 18. Červenec 2011.] http://martinfowler.com/articles/continuousIntegration.html.

**Humble, Jez a Farley, David. 2010.** *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Upper Saddle River, NJ : Addison-Wesley, 2010. ISBN: 9780321601919.

**Menšík, Vlastimil. 2009.** CI za použití Hudsonu a Mavenu. *ET NETERA.* [Online] ET NETERA, 20. říjen 2009. [Citace: 27. červenec 2011.] http://www.etnetera.cz/cz/21447-tech\_streda/ci\_za\_pouziti\_hudsonu\_a\_mavenu.html.

**Wikipedia contributors. 2011.** Continuous integration. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia Foundation, 20. červenec 2011. [Citace: 27. červenec 2011.] http://en.wikipedia.org/w/index.php?title=Continuous\_integration&oldid=440518476.

**Zikmund, Štěpán. 2011.** Diplomová práce: Deployment aplikací v PHP. Praha : KIT FIS VŠE, 2011.

1. QA = quality assurance – kontrola kvality [↑](#footnote-ref-1)
2. Volně podle [Wikipedia contributors, 2011] [↑](#footnote-ref-2)
3. Kontinuální (nebo také průběžná) integrace [↑](#footnote-ref-3)
4. Přeloženo z [Fowler, 2006] [↑](#footnote-ref-4)
5. Viz http://www.extremeprogramming.org/rules/integrateoften.html [↑](#footnote-ref-5)
6. Zkracováno jako EP - viz [Beck, 2005] [↑](#footnote-ref-6)
7. SCM - Source Code Management Tools [↑](#footnote-ref-7)
8. Viz http://subversion.tigris.org/ [↑](#footnote-ref-8)
9. Concurrent Versions System – viz http://savannah.nongnu.org/projects/cvs [↑](#footnote-ref-9)
10. Viz https://jazz.net/projects/rational-team-concert/ [↑](#footnote-ref-10)
11. Viz http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview [↑](#footnote-ref-11)
12. Viz http://git-scm.com/ [↑](#footnote-ref-12)
13. Viz http://mercurial.selenic.com/ [↑](#footnote-ref-13)
14. Viz http://bazaar.canonical.com/ [↑](#footnote-ref-14)
15. Viz http://www.bitkeeper.com/ [↑](#footnote-ref-15)
16. Viz http://kerneltrap.org/node/4966 [↑](#footnote-ref-16)
17. Získání kopie všech souborů z repositáře [↑](#footnote-ref-17)
18. Tzv. build [↑](#footnote-ref-18)
19. Java Development Kit http://www.oracle.com/technetwork/java/javase/downloads/index.html [↑](#footnote-ref-19)
20. Integrated development environment neboli Vývojové prostředí [↑](#footnote-ref-20)
21. Viz http://www.gnu.org/software/make/ [↑](#footnote-ref-21)
22. Viz http://ant.apache.org/ [↑](#footnote-ref-22)
23. Viz http://www.phing.info [↑](#footnote-ref-23)
24. TDD neboli vývoj řízený testy [Beck, 2002] [↑](#footnote-ref-24)
25. Viz http://www.martinfowler.com/bliki/Xunit.html [↑](#footnote-ref-25)
26. Viz http://www.junit.org/ [↑](#footnote-ref-26)
27. Viz http://www.nunit.org/ [↑](#footnote-ref-27)
28. Viz http://www.phpunit.de/ [↑](#footnote-ref-28)
29. Je důležité, aby integraci neprováděl na svém počítači, protože například pokud zapomene uložit nějaký soubor do úložiště, sestavení by prošlo (soubor má k dispozici), ale u ostatních členů týmu by selhalo (neměli by k dispozici chybějící soubor). [↑](#footnote-ref-29)
30. Viz kapitola 2.3 [↑](#footnote-ref-30)
31. Viz http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBuildsInTrouble.rdoc [↑](#footnote-ref-31)
32. http://dattein.com/blog/arduino-build-light/ [↑](#footnote-ref-32)
33. http://fabiopereira.me/blog/2009/12/15/build-dashboard-radiator-your-build-light-2/ [↑](#footnote-ref-33)
34. Viz <http://cruisecontrol.sourceforge.net/> [↑](#footnote-ref-34)
35. Viz <http://phpundercontrol.org/> [↑](#footnote-ref-35)
36. Viz <http://code.google.com/p/xinc/> [↑](#footnote-ref-36)
37. Viz <http://sismo.sensiolabs.org/> [↑](#footnote-ref-37)
38. Viz <http://travis-ci.org/> [↑](#footnote-ref-38)
39. Viz <http://www.atlassian.com/software/bamboo/overview> [↑](#footnote-ref-39)
40. Viz <http://www.atlassian.com/> [↑](#footnote-ref-40)
41. Viz <http://news.cnet.com/8301-30685_3-20000019-264.html> [↑](#footnote-ref-41)
42. Viz <http://jenkins-ci.org/content/whos-driving-thing> [↑](#footnote-ref-42)
43. Viz <http://jenkins-ci.org/content/hudsons-future> [↑](#footnote-ref-43)
44. Viz <http://jenkins-ci.org/content/jenkins>, <http://jenkins-ci.org/> [↑](#footnote-ref-44)
45. Viz <http://kitscm.vse.cz/> [↑](#footnote-ref-45)
46. Viz http://www.zend.com/en/products/studio/ [↑](#footnote-ref-46)
47. Viz http://eclipse.org/pdt/ [↑](#footnote-ref-47)
48. Viz http://netbeans.org/features/php/ [↑](#footnote-ref-48)
49. Viz [Bergmann, 2008] [↑](#footnote-ref-49)
50. Viz http://blueparabola.com/blog/subversion-commit-hooks-php [↑](#footnote-ref-50)
51. Viz http://phpadvent.org/2008/dont-commit-that-error-by-travis-swicegood [↑](#footnote-ref-51)
52. Viz http://framework.zend.com/ [↑](#footnote-ref-52)
53. Viz https://github.com/sebastianbergmann/phpdcd [↑](#footnote-ref-53)
54. Viz https://github.com/sebastianbergmann/phpdcd/issues/5 [↑](#footnote-ref-54)
55. Viz https://github.com/sebastianbergmann/phpdcd#readme [↑](#footnote-ref-55)
56. Viz https://github.com/sebastianbergmann/phpdcd/issues/6 [↑](#footnote-ref-56)
57. Viz https://github.com/sebastianbergmann/phpcpd [↑](#footnote-ref-57)
58. Viz https://github.com/sebastianbergmann/phploc [↑](#footnote-ref-58)