VYSOKÁ ŠKOLA EKONOMICKÁ V PRAZE

Fakulta informatiky a statistiky

Katedra informačních technologií

**BAKALÁŘSKÁ PRÁCE**

2012 Martin Hujer

**Vysoká škola ekonomická v Praze**

**Fakulta informatiky a statistiky**

**Katedra informačních technologií**

Studijní program: Aplikovaná informatika

Obor: Informatika

**Kontinuální integrace při vývoji webových aplikací v PHP**

**BAKALÁŘSKÁ práce**

Student : Martin Hujer

Vedoucí : Ing. Jan Mittner

Oponent : @tdo Jméno a příjmení s tituly

**2012**

**Prohlášení**

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

V Praze dne @tdo den. měsíc 2012 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Martin Hujer

**Poděkování**

Zde je prostor pro vyjádření poděkování… @tdo ………………………….

**Abstrakt**

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.

Obsahuje zaměření a hlavní cíl práce, způsob dosažení cíle, přínos práce (vlastní příspěvek k řešenému tématu) a stručně popsanou strukturu práce. @tdo

**Klíčová slova**

Seznam nejvýznamnějších odborných výrazů charakterizujících téma závěrečné práce. @tdo

**Abstract**

Anglická verze abstraktu. @tdo

**Keywords**

seznam klíčových slov v anglickém jazyce. @tdo

@tdo vygenerovat obsah //až ke konci, jinak má Word problémy

# Úvod

@todo Webové aplikace se dnes rozvíjejí a mění rychle, takže je potřeba sledovat a kontrolovat kvalitu kódu.

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 zařazují fáze integrace, testování a kontroly kvality až po dokončení fáze vývoje (viz **Chyba! Nenalezen zdroj odkazů.**). Doba samotného vývoje software je často odhadnuta nesprávně, i když tomu lze částečně předejít. [MCCONNELL, 2006]

Obrázek 1: Schéma vodopádového modelu vývoje software

//@todo najít knížky od Voříška/Buchalcevové a z nich vzít obrázek a citaci waterfallu a Scrumu

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. Kontinuální integrace je často využívána současně s tzv. agilními metodikami vývoje, kdy je software vyvíjen v krátkých iteracích.

//@todo tady bude schéma scrumu

# Rešerše zdrojů pojednávajících o kontinuální integraci

Prvním zdrojem zabývajícím se kontinuální integrací byl [FOWLER, 2006]. Většina dalších prací, které se kontinuální integrací zabývají, cituje jeho původní definici. Mnoho informací o kontinuální integraci shrnul [DUVALL, 2007] (autorem předmluvy je již výše zmíněný Fowler). Oba tyto zdroje se zabývají kontinuální integrací obecně, případně pro projekty vyvíjené v jazyce Java.

Důležitým zdrojem byl [BERGMANN, 2011], který se zabývá kvalitou kódu a testováním aplikací v jazyce PHP.

Dalším informačním zdrojem je web <http://jenkins-php.org/>, který obsahuje ukázkovou šablonu pro nastavení kontinuální integrace pro PHP projekty na integračním serveru Jenkins.

V průběhu mé práce vyšla kniha [SMART, 2011], která se detailně zabývá instalací, konfigurací a správou integračního serveru Jenkins.

Dále vyšla kniha [BERGMANN, 2011b], která se zabývá instalací a konfigurací integračního serveru Jenkins přímo pro webové aplikace v PHP. Autor nicméně zvolil některé jiné nástroje než já.

# Charakteristika konceptu kontinuální integrace

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:

Kontinuální integrace 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.[[3]](#footnote-3)

Kontinuální integrace byla poprvé popsána[[4]](#footnote-4) jako jedna z praktik tzv. Extrémního programování[[5]](#footnote-5), nicméně je ji samozřejmě možné využít i bez implementace ostatních praktik EP.

## Přehled technik a zásad kontinuální integrace

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, 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[[6]](#footnote-6), 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[[7]](#footnote-7) nebo dnes již zastaralé CVS[[8]](#footnote-8). Z komerčních lze jmenovat Rational Team Concert[[9]](#footnote-9) od IBM nebo Team Foundation Server[[10]](#footnote-10) od společnosti Microsoft. Mezi nejznámější open-source nástroje pracující s distribuovaným úložištěm patří Git[[11]](#footnote-11) (je používaný pro verzování jádra operačního systému Linux), Mercurial[[12]](#footnote-12) nebo Bazaar[[13]](#footnote-13). Z komerčních lze uvést BitKeeper[[14]](#footnote-14) od společnosti BitMover, Inc. (používaný pro verzování jádra operačního systému Linux v letech 2002-2005[[15]](#footnote-15)).

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, 2006] doporučuje jednoduché pravidlo: Pokud na čistě nainstalovaném počítači provedete checkout[[16]](#footnote-16) repositáře, tak by mělo být možné provést sestavení aplikace[[17]](#footnote-17). Výjimku tvoří například operační systém, JDK[[18]](#footnote-18) 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[[19]](#footnote-19).

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[[20]](#footnote-20) používaný v operačních systémech Linux a Unix, Ant[[21]](#footnote-21) používaný pro projekty psané v jazyce Java nebo Phing[[22]](#footnote-22) 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 sestavení 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 jazyce 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[[23]](#footnote-23), 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[[24]](#footnote-24) frameworky, kde x značí programovací jazyk. Existuje tedy JUnit[[25]](#footnote-25) pro Javu, NUnit[[26]](#footnote-26) pro .NET, PHPUnit[[27]](#footnote-27) pro PHP a další.

Testy jsou většinou sdružovány do sad testů a po jejich proběhnutí je vygenerována zpráva, která informuje o tom, zda některé testy selhaly.

[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 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[[28]](#footnote-28). 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 sestavení a autorovi zašle e-mail s výsledkem.

Integrační server není nezbytně nutný, ale jeho použití je pohodlnější. Jeho další výhodou je, že pokud součástí sestavení je i statická analýza zdrojového kódu[[29]](#footnote-29), je možné na výstupu integračního serveru sledovat dlouhodobé trendy v kvalitě kódu.

### Sestavení musí být rychlé

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 sestavení netrvalo déle než 10 minut. Pokud by trval déle, výrazně se zvyšuje šance, že ho vývojáři nebudou spouštět lokálně.

Často je sestavení zpomalováno testy, které pracují s externími zdroji, jako je databáze nebo služby. V takovém případě je možné sestavení rozdělit na dvě části – primární (spouští se po uložení změn 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 sestavení trvalo příliš dlouho, 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 oddělovači 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 rozšíření do IDE. Týmy často experimentují s vizuálními pomůckami, které ukazují stav posledního sestavení. Některé používají například červenou a zelenou lávovou lampu[[30]](#footnote-30), LED diody řízené Arduinem[[31]](#footnote-31) nebo detailnější tabuli s přehledem stavu jednotlivých sestavení[[32]](#footnote-32).

### Automatizované nasazení

Využívání kontinuální integrace umožňuje častěji získávat stabilní sestavení aplikace, čehož lze využít pro zvýšení frekvence vydávání a nasazování nových verzí. Tato technika se označuje jako Continuous Delivery. Pokud by systém automaticky nasadil do provozu každé úspěšné sestavení, které projde automatizovanými testy, označovali bychom to jako Continuous Deployment.

Podrobný rozbor tématu však přesahuje rozsah této práce. Více informací lze nalézt například v [HUMBLE, 2010], případně v [ZIKMUND, 2011] se zaměřením jen na PHP.

## Typický průběh práce vývojáře v prostředí kontinuální integrace

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 týmu
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

## Testování software

Jak již naznačuje kapitola 3.1.3, nedílnou součástí kontinuální integrace by mělo být testování software.

Testování je neustálý proces vývoje, využívání a udržování testovacích nástrojů za účelem měření a zlepšení kvality testovaného software. [CRAG, 2002]

Testování software lze rozdělit do několika úrovní, které lze znázornit pomocí tzv. „V“ Modelu (viz Obrázek 2). V dolní části obrácené pyramidy převládá automatizované testování, v horní části naopak manuální.



Obrázek 2: "V" model testování software [CRAG, 2002]

### Manuální testování

Při manuálním testování procházejí testeři aplikaci podle instrukcí v testovacích scénářích a ověřují, zda chování a výsledky aplikace odpovídají očekávaným. Manuálním testováním webových aplikací se tato práce nezabývá.

Nicméně manuální testování může využívat výhod kontinuální integrace například tak, že každé úspěšné sestavení, které projde automatizovanými testy, bude automaticky zpřístupněno k manuálnímu testování na testovacím serveru.

### Automatizované testování

Automatizované testování funkčnosti software většinou probíhá tak, že dochází k automatizovanému porovnávání očekávaných výstupů aplikace pro dané vstupy s aktuálními výstupy. Může probíhat na úrovni tříd (tzv. jednotkové testování), na úrovni komponent (integrační testování) nebo na úrovni celého systému (systémové).

### Metriky – výsledky automatizovaných testů, pokrytí kódu testy

V kontextu kontinuální integrace jsou automatizované testy velmi užitečné. Díky tomu, že sestavení projektu (včetně spuštění automatizovaných testů) probíhá po každé změně zdrojových kódů, je v případě selhávajících testů snadné odhalit, která změna to způsobila.

Zjištění, že zda nějaké testy selhávají, je většinou rozhodující pro další průběh sestavení. V takovém případě nemá smysl provádět například kontrolu dodržování standardů pro psaní kódu. Lze zjednodušeně říci, že pokud se některá část aplikace nechová podle očekávání, tak už informace, že některá místa v kódu nejsou elegantně formátovaná, není tak užitečná.

#### Míra pokrytí kódu testy

V souvislosti s automatizovaným testováním se často získává metrika Míra pokrytí kódu testy (code coverage). Tato metrika ukazuje, které řádky kódu byly provedeny během testování. Je možné si to snadno ukázat na příkladu Kód 3.1. Pokud funkce checkSomething() bude z testů zavolána jen s parametrem true, tak bude výsledné pokrytí kódu testy 50%.

function checkSomething($condition) {

if ($condition) {

return true;

} else {

return false;

}

}

Kód 3.1 Míra pokrytí kódu testy

Je důležité si uvědomit, že i když je řádek během testování zavolán, tak to neznamená, že nemůže obsahovat chyby. Dobře to ilustruje [WOOD, 2008], kdy v kódu, který má 100% pokrytí testy postupně odhalí několik zásadních chyb. Dále mluví o tom, že není nutné se hnát za co nejvyšším pokrytím kódu testy:

Nicméně, je důležité si pamatovat, že cílem není dosáhnout 100% pokrytí, ale mít úplné a komplexní jednotkové testy. [WOOD, 2008]

Metriku lze ovšem využít na zjištění, které části aplikace jsou testovány výrazně méně než ostatní.

## 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. Platí zde známé:

Co se nedá měřit, to se nedá řídit. [DEMARCO, 1982]

Sledování a vyhodnocování metrik zdrojového kódu je důležité, protože mohou pomoci odhalit tzv. zatuchlý kód (code smell), který může způsobovat komplikace v pozdějších fázích softwarového projektu. O tom, jak psát kvalitní a čistý programový kód detailně pojednává například [MARTIN, 2008].

Mezi nejčastěji získávané metriky patří:

* Počet řádků kódu, programových struktur (viz kapitola 3.4.1)
* Složitost (komplexita) kódu (viz kapitola 3.4.2)
* Dodržování Coding Standards (standardů pro psaní kódu, viz kapitoly 3.4.1a 3.4.3)
* Duplicitní kód (viz kapitola 3.4.4)

### Počet řádků kódu, programových struktur

Zjišťování počtu řádků kódu a jednotlivých programových struktur (třídy, metody, rozhraní, …) může být v rámci kontinuální integrace zajímavé například pro sledování trendů. Nicméně je důležité, aby si řídící pracovníci uvědomili, že produktivitu vývojáře není možné měřit podle přidaných řádků kódu. Dobře to vysvětluje [MCCONNELL, 2008]:

Pokud je produktivita měřena počtem řádků kódu, znamená to, že vývojář, který na vyřešení problému použije 10x více kódu je produktivnější než vývojář, který napíše desetinu kódu. To jasně nedává smysl. [MCCONNELL, 2008]

Případně to lze ukázat na jiném příkladu. Vývojář provede refaktoring a vyčlení duplicitní kód do samostatné třídy, čímž sníží celkový počet řádků kódu (kód bude do budoucna lépe udržovatelný). Pokud by produktivita vývojáře byla měřena jen podle řádků kódu, tak i přes nezpochybnitelný přínos by jeho produktivita byla dokonce záporná.

Data jsou spíše podkladem pro měření komplexity kódu (viz následující kapitola).

### Složitost (komplexita) kódu

V průběhu vývoje softwarového projektu je vhodné sledovat složitost jednotlivých částí zdrojového kódu. Vysoká složitost kódu může znamenat nižší přehlednost a často znesnadňuje testování kódu. Pro dlouhodobou udržitelnost rozvoje softwarového projektu je vhodné složitost sledovat a případně části kódu refaktorovat (změna struktury zdrojového kódu při zachování funkcionality).

#### Cyklomatická složitost

Pojem cyklomatická složitost (označována jako CCN), tak jak ji popsal [MCCABE 1976], vychází z teorie grafů. Zjednodušeně lze říci, že jde o celkový počet míst, kde se kód větví (typicky podmínky nebo cykly).

#### Rozšířená cyklomatická složitost

Rozšířená cyklomatická složitost (CCN2) započítává jako další rozvětvení i boolovské výrazy v podmínkách. Některé nástroje tuto uvádějí jako běžnou cyklomatickou složitost.

#### Rizika vysoké cyklomatické složitosti

Tabulka 1: Cyklomatická složitost a riziko spolehlivosti [MCCABE JR., 2008]

|  |  |
| --- | --- |
| **Cyklomatická složitost** | **Riziko spolehlivosti** |
| 1 – 10 | Jednoduchá metoda, malé riziko |
| 11- 20 | Složitější, střední riziko |
| 21 – 50 | Složitá, vysoké riziko |
| >50 | Netestovatelné, VELMI VYSOKÉ RIZIKO |

Tabulka 2: Cyklomatická složitost a pravděpodobnost chybné opravy [MCCABE JR., 2008]

|  |  |
| --- | --- |
| **Cyklomatická složitost** | **Pravděpodobnost chybné opravy** |
| 1 – 10 | 5% |
| 20 –30 | 20% |
| > 50 | 40% |
| Blížící se 100 | 60% |

#### NPath složitost

NPath složitost, tak jak ji definoval [NEJMEH 1988], označuje počet možných průchodů blokem zdrojového kódu. Autor doporučuje hodnotu NPath složitosti sledovat a pokud překročí 200, tak konkrétní úsek zdrojového kódu prověřit a případně ho refaktorovat.

### Standardy pro psaní kódu

Na softwarových projektech většinou spolupracuje více vývojářů, z nichž každý může být zvyklý zapisovat zdrojový kód jiným způsobem. Pokud by i na společném projektu zapisovali kód různými styly, měli by ostatní členové týmu problémy mu snadno porozumět.

Tento problém řeší vytvoření standardů pro psaní kódu, které definují, jak má být zdrojový kód formátován. Většinou jde o dokument přístupný všem členům týmu. Skládá se z několika částí:

* pravidla formátování samotného souboru (odsazování kódu, maximální délky řádků, znak ukončení řádku)
* jmenné konvence – jak pojmenovávat třídy, rozhraní, funkce, metody, proměnné a konstanty, …
* styl vlastního psaní – jak zapisovat jednotlivé programové struktury, kam umístit závorky uvozující jednotlivé části (třídy, metody, …), jak zapisovat řídící struktury (podmínky, cykly, …)
* dokumentační standard

Výsledkem je poté konzistentní, snadno čitelný a pochopitelný kód. Není možné na první pohled rozpoznat, který vývojář psal danou část kódu. To je důležité zejména při uplatňování jedné z praktik extrémního programování, tzv. Collective code ownership (společného vlastnictví kódu).

Samozřejmě není nutné pro každý projekt vytvářet nové standardy, což ani není žádoucí. Mnohem vhodnější je zvolit již existující a zavedený standard pro formátování (například Code Conventions for the Java TM Programming Language[[33]](#footnote-33), PEAR Coding Standars[[34]](#footnote-34) nebo Zend Framework Coding Standard for PHP[[35]](#footnote-35)). K jejich výhodám patří, že vývojáři už na ně budou pravděpodobně přivyklí a také podpora v různých nástrojích (jak editorech, tak nástrojích pro analýzu kódu).

O tom, proč je prakticky nutností dodržovat nějaké standardy pro psaní kódu, dále pojednává například [MYTTON, 2004] nebo [MARTIN, 2007].

### Duplicitní kód

Duplicitní kód se v softwarových projektech může objevit například tak, že vývojář potřebuje metodu s velmi podobnou funkcionalitou, která už je obsažena v existující, takže ji jen zkopíruje a drobně upraví. Kvůli tomu může v budoucnu dojít k problémům, kdy je upraven jen jeden výskyt tohoto kódu a ne všechny jeho kopie.

Řešením v kontinuální integraci je nasazení nástroje, který tato místa vyhledává.

## Přínosy kontinuální integrace (//@todo Duval 64 + Ch3)

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

### Snížení rizik

* chyby jsou objeveny a opraveny rychleji
  + možnost využití http://martinfowler.com/bliki/DiffDebugging.html
* lze měřit kvalitu software

### Snížení množství manuálních činností

### Nasaditelný SW je kdykoliv k dispozici

### Viditelnost projektu

### Větší důvěra týmu v kvalitu

# Výběr vhodného integračního serveru a jeho konfigurace

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

## Přehled trhu serverů pro kontinuální integraci

Na trhu jsou k dispozici různé servery pro kontinuální integraci. Ten nejstarší, CruiseControl[[36]](#footnote-36), byl vyvinut ve společnosti ThoughtWorks, pro kterou pracuje Martin Fowler, autor známého článku [FOWLER, 2006]. Existuje pro něj rozšíření phpUnderControl[[37]](#footnote-37), které usnadňuje prvotní nastavení nástrojů pro kontinuální integraci v PHP, nicméně již není aktivně vyvíjeno, takže jeho nasazení bych spíše nedoporučoval. Velmi známým integračním serverem je Jenkins, který je detailně popsán v kapitole 4.2. Kromě toho existují další méně známé integrační servery, například Xinc[[38]](#footnote-38), který je napsaný v jazyce PHP.

Z komerčních serverů pro kontinuální integraci bych zde uvedl Bamboo[[39]](#footnote-39) od společnosti Atlassian[[40]](#footnote-40). Výhodou tohoto serveru je snadné propojení s dalšími nástroji pro projektové řízení od této společnosti. Dalším komerčním serverem je uBuild[[41]](#footnote-41), který je nástupcem známého serveru AnthillPro. V neposlední řadě bych uvedl integrační server zaměřený především na jazyk Java a .NET, TeamCity[[42]](#footnote-42), jehož tvůrcem je původně česká společnost JetBrains[[43]](#footnote-43). Často používaný je také Team Foundation Server[[44]](#footnote-44) od společnosti Microsoft. Nejedná se jen o integrační server, ale spíše platformu zastřešující celý proces vývoje software. Je zaměřený především na vývoj ve frameworku .NET a integraci s vývojovým prostředím Visual Studio.

### Příbuzný software

S integračními servery úzce souvisí i další software, který je může doplňovat (ale funguje nezávisle na nich). Zařadil bych sem Sismo[[45]](#footnote-45), což je Continuous Testing Server – zaměřuje se jen na spouštění testů. Díky tomu ho lze využít jako komplement k integračnímu serveru pro rychlé spouštění jednotkových testů lokálně, po commitu do repositáře ve verzovacím systému GIT. Podobně funguje i Travis CI[[46]](#footnote-46), jehož cílem je vytvořit distribuovaný systém pro spouštění testů open-source software.

Sonar[[47]](#footnote-47) je open-source nástroj na řízení kvality zdrojových kódů. V základu se zaměřuje na jazyk Java, ale pomocí open-source nebo i komerčních rozšíření je do něj možné doplnit podporu pro další jazyky (C, C#, PHP, …). Je možné ho využít ve spojení s různými servery pro kontinuální integraci jako je Hudson nebo Jenkins[[48]](#footnote-48).

## Jenkins / Hudson

Velmi známým integračním serverem je Jenkins (resp. Hudson). Někdy dochází k zaměňování těchto názvů, proto je si je 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[[49]](#footnote-49) společnost Sun Microsystems, čímž mj. získala právo na využívání názvu Hudson, který si později registrovala jako ochrannou známku. Oracle se během sjednocování infrastruktury s ostatními open-source projekty, které zastřešují, dostal do sporu[[50]](#footnote-50) s komunitou vývojářů Hudsonu a ti se rozhodli zbavit závislosti na společnosti Oracle[[51]](#footnote-51) a začali Hudson dále vyvíjet pod názvem Jenkins[[52]](#footnote-52).

Společnost Oracle nadále 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 serveru GitHub.com {rozepsat info o něm}, takže je možné snadno provést srovnání rychlosti vývoje a zájmu vývojářů o ně:

Tabulka 3: Srovnání aktivity u projektů Hudson a Jenkins na serveru GitHub.com

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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. Je to zajímavá ukázka toho, že vývoj software komunitou dobrovolníků může postupovat dopředu rychleji než vývoj software, který zaštiťuje velká společnost.

Při výuce vývoje v Javě na VŠE se dříve využíval integrační server Hudson, nicméně dnes (25. 2. 2012) již to je Jenkins[[53]](#footnote-53), což potvrzuje moji domněnku, že Jenkins je vhodnější volbou než Hudson. Vzhledem k tomu, že většina zdrojů (viz kapitola 2), které se kontinuální integrací v prostředí PHP zabývají, zvolila jako integrační server Jenkins, nenašel jsem důvod, proč vybrat jiný.

## Instalace serveru Jenkins

Pro ukázkovou instalaci integračního serveru jsem zvolil aktuální verzi serveru Jenkins (k 8. 3. 2012 1.454). Ty vycházejí velmi často (většinou každý týden), což není být vhodné pro instalace, které musí být stabilní. Nasazení nové verze s sebou přináší nutnost otestovat, zda skript pro sestavení a veškerá rozšíření fungují jako dříve. Jenkins proto zavedl tzv. LTS verze (Long-Term Support)[[54]](#footnote-54), které jsou založeny na některé starší verzi, která se osvědčila a jsou do nich zpětně zahrnovány jen opravy chyb a ne nová funkcionalita.

Jenkins je k dispozici jak ve formátu Java Web Archive (.war), tak i ve formě instalačních balíčků pro operační systém Windows, a běžné linuxové distribuce. Pro některé linuxové distribuce jsou k dispozici i úložiště balíčků[[55]](#footnote-55), takže je možné Jenkins snadno aktualizovat na novou verzi.

Instalaci začneme stažením balíčku ve formátu .war, který umístíme do vybrané složky (například I:/BP-jenkins/). Jenkins si ve výchozím nastavení ukládá nastavení a soubory do domovského adresáře aktuálně přihlášeného uživatele. Vhodnější je data ukládat do nějakého jiného adresáře, což lze nastavit do hodnoty proměnné prostředí JENKINS\_HOME. Samotný Jenkins lze poté spustit jako běžnou aplikaci v Javě pomocí příkazu java -jar jenkins.war.

Než zadávat tyto příkazy při každém spouštění Jenkinsu ručně, je vhodnější vytvořit spouštěcí dávkový soubor. Pojmenovat ho lze například jako jenkins-start.cmd a obsah bude tento:

set JENKINS\_HOME=i:\BP-jenkins\data\

java -jar jenkins.war

Po jeho spuštění začne startovat Jenkins. Z výpisu je zřejmé, že se použil námi určený adresář:

Jenkins home directory: i:\BP-jenkins\data found at: EnvVars.masterEnvVars.get("JENKINS\_HOME")

Posledním řádkem výpisu by měla být informace oznamující, že spuštění proběhlo v pořádku:

INFO: Jenkins is fully up and running

Ověřit to lze otevřením URL http://localhost:8080/ ve webovém prohlížeči. Měla by se zobrazit hlavní stránka serveru Jenkins (viz Obrázek 3). Tím je dokončena základní instalace a je možné začít vytvářet projekt.

@todo Jenkins za Apache proxy?

@todo zabezpečení Jenkinse



Obrázek 3: Hlavní stránka Jenkinsu po prvním spuštění

## Možnosti konfigurace serveru Jenkins

Jedna z možností, jak ovládat a konfigurovat Jenkins, je přes webové rozhraní, jak je naznačeno výše. Další možností je pomocí klienta příkazové řádky. Jeho výhodou je možnost snadno spustit více příkazů za sebou, bez nutnosti manuálního procházení webových stránek. Nevýhodou je nemožnost ovládat vše, co je možné z webového rozhraní. Potřebnou verzi CLI klienta lze snadno získat přímo ze serveru Jenkins pomocí příkazu:

wget http://localhost:8080/jnlpJars/jenkins-cli.jar

Bylo možné si všimnout, že se Jenkins automaticky načetl v jazyce, který je nastavený jako výchozí ve webovém prohlížeči. S největší pravděpodobností to tedy byla čeština. Český překlad Jenkinsu není kompletní, což by znepříjemňovalo práci a zároveň mohlo být v některých situacích matoucí. Lze to vyřešit nastavením webového prohlížeče tak, aby požadoval webové stránky v angličtině, což by museli provést všichni, kteří budou Jenkins využívat. Vhodnějším řešením je instalace rozšíření Locale[[56]](#footnote-56), které umožňuje nastavit jazyk napevno, bez ohledu na konfiguraci webového prohlížeče. Rozšíření nainstalujeme takto:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin locale

Server je po instalaci rozšíření vždy nutné restartovat, což lze provést příkazem:

java -jar jenkins-cli.jar -s http://localhost:8080 safe-restart

(Nicméně pokud je server spuštěný je pomocí java -jar jenkins.war, tak restart nelze provést a je nutné server ukončit a poté opět spustit.)

Rozšíření už je aktivní, takže je možné přejít do nastavení systému[[57]](#footnote-57) a sekci „Locale“ upravit takto:



Obrázek 4: Jenkins - nastavení anglického prostředí

Z historických důvodů Hudson a Jenkins používají pro označení úspěšných sestavení modrou barvu (resp. modré kuličky), což nemusí být na první pohled zřejmé (uživatelé spíše čekají zelenou barvu). Lze to vyřešit instalací rozšíření Green Balls[[58]](#footnote-58).

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin greenballs

# Přehled nástrojů pro kontinuální integraci v 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 instalaci, automatizaci, tvorbu dokumentace, apod.

## PEAR

PEAR[[59]](#footnote-59) neboli PHP Extension and Application Repository je systém pro distribuci a instalaci balíčků komponent pro jazyk PHP. Tento balíčkovací systém je využit v dalších kapitolách pro instalaci jednotlivých nástrojů. Detailní popis instalace samotného systému PEAR přesahuje rozsah této práce. Instalační instrukce pro jednotlivé operační systémy lze nalézt na <http://pear.php.net/manual/en/installation.php>. Pro správné pochopení použití PEAR je nutné definovat základní pojmy. Jsou to tyto: balíček a kanál.

Balíček lze popsat jako množinu souborů a složek, které tvoří logický celek (komponentu). Součástí balíčku jsou zároveň další metainformace, jako je jeho verze a závislosti na dalších balíčcích.

Kanál je běžná webová stránka, která navíc obsahuje metainformace o balíčcích, které jsou tam k dispozici (informace jsou umístěny v souboru channel.xml).

PEAR po instalaci obsahuje jen oficiální kanál pear.php.net. Pro instalaci balíčků z jiných umístění je nutné nejprve další kanály inicializovat. To lze provést pomocí příkazu pear channel-discover.

> pear channel-discover pear.phpunit.de

Adding Channel "pear.phpunit.de" succeeded

Discovery of channel "pear.phpunit.de" succeeded

Poté už je možné instalovat jednotlivé balíčky i z tohoto kanálu. V následujícím příkladu je phpunit označení pro PEAR kanál a phpcpd je konkrétní balíček:

> pear install phpunit/phpcpd

Starting to download phpcpd-1.3.5.tgz (8,746 bytes)

.....done: 8,746 bytes

install ok: channel://pear.phpunit.de/phpcpd-1.3.5

Tento základní popis funkčnosti by měl být dostatečný pro využití PEARu pro instalaci balíčků potřebných pro zavedení kontinuální integrace. Detailní informace o funkčnosti nástroje PEAR lze nalézt v oficiální dokumentaci: <http://pear.php.net/manual/en/>.

## PHP Lint

Kontrolu kvality kódu nemá smysl provádět dříve, než si jsme jisti, že napsaný kód je syntakticky v pořádku (to, že kód nelze zkompilovat je větší problém než nedodržování stanovených Coding Standards). Zároveň by kvůli tomu mohly nástroje pro kontrolu kvality vracet nepřesné výsledky.

Některá moderní IDE jako Zend Studio[[60]](#footnote-60), Eclipse PDT[[61]](#footnote-61) nebo Netbeans[[62]](#footnote-62) už sice kontrolu syntaktických chyb obsahují (viz Obrázek 5), ale slouží spíše jako upozornění pro vývojáře. Integrační nástroj nemůže spoléhat na to, že si vývojář chyby všimne a opraví ji.



Obrázek 5: 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 je kontrola spuštěna na skriptu test1-error.php, který obsahuje syntaktickou chybu, vrátí PHP chybový výstup (viz Kód 5.2)

<?php

echo "Kontinuální integrace" //chybí středník

Kód 5.1 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 5.2 Chybový výstup z PHP Lint

Pokud je soubor syntakticky v pořádku (viz Kód 5.3), vrátí PHP Lint výstup (Kód 5.4)

<?php

echo "Kontinuální integrace";

Kód 5.3 Zdrojový kód souboru test2-ok.php

> php -l test2-ok.php

No syntax errors detected in test2-ok.php

Kód 5.4 Výstup z PHP Lint, když je kód v pořádku

### Kontrola dopředné kompatibility

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

Při provedení kontroly pomocí PHP Lint (verze PHP 5.3) na souboru (viz Kód 5.5), je vše v pořádku (Kód 5.6).

<?php

class Trait {}

Kód 5.5 test3-trait.php

>php -l test3-trait.php

No syntax errors detected in test3-trait.php

Kód 5.6 Výstup z PHP Lint (verze PHP 5.3) na souboru test3-trait.php

Pokud je ovšem použito PHP ve verzi 5.4.0alpha2, kde trait je klíčovým slovem, vrátí chybový výstup (viz Kód 5.7)

> 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 5.7 Chybový výstup z PHP Lint (verze PHP 5.4.0alpha2) na souboru test3-trait.php

Pro účely kontinuální integrace 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[[63]](#footnote-63), je možné 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 využít pre-commit hook, který může probíhající commit zrušit. Lze využít už hotové skripty pro SVN[[64]](#footnote-64) nebo GIT[[65]](#footnote-65). 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[[66]](#footnote-66) využívá jako šablonovací jazyk přímo PHP a šablony jsou ve výchozím nastavení uloženy v souborech s příponou PHTML (viz Kód 5.8). 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 5.8 Příklad šablony v PHTML souboru (test4-template.phtml)

## PHP\_CodeSniffer

PHP\_CodeSniffer[[67]](#footnote-67) je nástroj určený na analýzu zdrojových kódů aplikace a kontrolu, zda odpovídají zvoleným standardům pro formátování souborů se zdrojovým kódem.

Je možné zvolit již nějaký předdefinovaný standard pro formátování (k dispozici jsou například standardy používané v knihovně PEAR, Zend Frameworku a dalších), případně je možné si vytvořit pravidla na míru pro svůj projekt.

Pro aplikace postavené nad Zend Frameworkem je samozřejmostí zvolit předdefinovaný standard. Nicméně provedl jsem jednu úpravu a upravil jsem pravidlo na omezení délky řádků zdrojového kódu. Výchozích 80 znaků je podle mého názoru přežitek z doby, kdy byla omezením šířka terminálu. Dnes, kdy je běžné vyvíjet software na monitorech s velkým rozlišením (fullHD – 1920x1080 bodů) nemá smysl omezovat délku řádku na méně než 120 znaků.

PHP\_CodeSniffer lze nainstalovat pomocí nástroje PEAR následujícím příkazem:

pear install PHP\_CodeSniffer

Pokud provedeme kontrolu dodržování standardu Zend[[68]](#footnote-68) na ukázkovém kódu, který porušuje několik pravidel:

<?php

class ArticlesController extends Zend\_Controller\_Action{

/\*\*

\* Index action

\*/

public function indexAction(){

$a = new ArticleModel(); }

}

Dostaneme výstup s přehledem chyb:

> phpcs --standard=Zend ArticlesController.php

FILE: php-cs\ArticlesController.php

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

FOUND 4 ERROR(S) AFFECTING 4 LINE(S)

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

2 | ERROR | Opening brace of a class must be on the line after the definition

6 | ERROR | Opening brace should be on a new line

7 | ERROR | Closing brace must be on a line by itself

8 | ERROR | Closing brace indented incorrectly; expected 0 spaces, found 4

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

Time: 0 seconds, Memory: 2.00Mb

PHP\_CodeSniffer je možné podobně jako PHP Lint spouštět jako pre-commit hook (a nedovolit do verzovacího systému uložit změny, které porušují coding standards). Nicméně při větším množství změněných souborů v rámci jednoho commitu může kontrola trvat dlouho, takže doporučuji provádět kontrolu dodržování coding standards až v rámci kontinuální integrace.

CodeSniffer samozřejmě podporuje výstup ve formátu vhodném pro zařazení do kontinuální integrace. Takový příkaz může vypadat například takto:

phpcs --standard=Zend --report=checkstyle --report-file=checkstyle-phpcs.xml .

## PHP CPD

PHP Copy/Paste Detector[[69]](#footnote-69) je nástroj, který odhalí zkopírované nebo duplicitní bloky kódu. Do systému ho lze nainstalovat pomocí příkazu

pear install phpunit/phpcpd

Jeho následné použití je jednoduché (viz Kód 5.9).

> 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 5.9 Ukázka použití PHP CPD

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

## PHPLOC

PHPLOC[[70]](#footnote-70) je jednoduchý nástroj, který umožňuje získat přehled o počtu řádků kódu v projektu a počítá některé další metriky.

Instalaci lze provést pomocí nástroje PEAR:

pear install phpunit/phploc

Jeho následné použití je jednoduché (viz Kód 5.10).

> 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 5.10 Ukázkový výstup z aplikace phploc

Samozřejmostí je opět výstup do formátu použitelného pro integrační server, který lze aktivovat přepínačem ‑‑log-csv <file>.

## PHP Depend

<http://pdepend.org/>

@todo Instalace

@spuštění

@co z něj do CI

pdepend --jdepend-chart=jdepend.svg --overview-pyramid=pyramid.svg --summary-xml=sum.xml ./models

- vysvětlit pyramidku

<http://pdepend.org/documentation/handbook/reports/overview-pyramid.html#ml06>

<http://pdepend.org/documentation/handbook/reports/abstraction-instability-chart.html>

## PHP MD

PHP MD (Mess Detector)[[71]](#footnote-71) je nástroj na analýzu zdrojových kódů v PHP a detekci potenciálně problematických míst ve zdrojových kódech. Snaží se být pro PHP podobným nástrojem, jako je PMD[[72]](#footnote-72) pro Javu. Je závislý na nástroji PDepend, jehož výstup používá jako vstup pro vlastní analýzu. Kontroluje jak přílišnou komplexitu kódu (viz kapitola 3.4.2), tak i některá další porušení standardů pro psaní kódu (viz 3.4.3).

Je vhodné vytvořit si vlastní XML soubor s pravidly[[73]](#footnote-73), která bude PHPMD ověřovat. Výhodou tohoto přístupu je, že pokud v budoucnu bude potřeba některá pravidla doplnit nebo upravit, nebude nutné upravovat skript pro sestavení. Soubor pravidel je typicky uložený v souboru phpmd.xml v kořenovém adresáři projektu.

Instalaci lze provést pomocí nástroje PEAR:

pear channel-discover pear.phpmd.org

pear channel-discover pear.pdepend.org

pear install --alldeps phpmd/PHP\_PMD

@todo

- krátké proměnné – ignoruje v cyklech apod. @todo

@co z něj do CI

## PHPUnit

PHPUnit[[74]](#footnote-74) je testovací framework pro jazyk PHP. Kromě podpory běžného jednotkového testování nabízí podporu také pro testování komponent závislých na databázi a testování pomocí nástroje Selenium[[75]](#footnote-75)

PHPUnit lze nainstalovat pomocí příkazů

pear config-set auto\_discover 1

pear install pear.phpunit.de/PHPUnit

Popis postupu vytváření testů s nástrojem PHPUnit přesahuje rozsah této práce, v následujících krocích se přepokládá, že jsou již vytvořené. Více informací o tvorbě testů pomocí PHPUnit lze nalézt například v [BERGMANN, 2005] nebo [BERGMANN, 2011].

Spuštění hotových testů je snadné. V adresáři, kde se nachází soubor phpunit.xml s konfigurací testů stačí zavolat příkaz phpunit a testy proběhnou podle nastavení v souboru phpunit.xml.

> phpunit

PHPUnit 3.6.10 by Sebastian Bergmann.

Configuration read from zf-tutorial\tests\phpunit.xml

..

Time: 1 second, Memory: 3.25Mb

OK (2 tests, 3 assertions)

Samozřejmostí je výstup ve formátu JUnit, se kterým umí pracovat integrační servery. Ten lze zajistit pomocí přepínače --log-junit phpunit-report.xml. Pomocí přepínače --coverage-clover lze z PHPUnit exportovat i statistiky pokrytí kódu testy. Opět jde o formát, který umí integrační servery zpracovávat.

V případě, že tým teprve začíná automatizované testování využívat, tak bych míru pokrytí kódu testy doporučoval nesledovat, neboť nízké hodnoty by mohly vývojáře spíše demotivovat.

## Generování API dokumentace

Při psaní zdrojového kódu je běžné do něj doplňovat tzv. dokumentační komentáře. Lze z nich totiž pak vygenerovat programátorskou dokumentaci (také nazývanou API dokumentace), kterou lze procházet pohodlněji než zdrojový kód.

Způsob zápisu používaný v jazyce PHP je založený na syntaxi používané v jazyce Java a nástroji Javadoc[[76]](#footnote-76), jen je upravený kvůli odlišnostem jazyka PHP.

<?php

/\*\*

\* Ukázková třída

\*

\* @author Martin Hujer

\*/

class MyClass

{

/\*\*

\* Funkce foo nic nedělá, jen vrátí předaný parametr.

\*

\* @param boolean $param Předaný parametr

\* @return boolean

\*/

public function foo($param)

{

return $param;

}  
}

Kód 5.11: Ukázka možnosti využití PHPDoc (soubor phpdoc01.php)

### PhpDocumentor

PhpDocumentor[[77]](#footnote-77) je původní nástroj na generování API dokumentace pro PHP skripty. Ovládá se pomocí příkazové řádky, takže ho není problém zařadit do skriptu pro sestavení. Jeho spuštění může vypadat například takto:

phpdoc -f phpdoc01.php -t doc

Obrázek 6 naznačuje, jak vypadá výsledná dokumentace ve výchozí HTML šabloně.



Obrázek 6: Ukázka dokumentace vygenerované pomocí skriptu PhpDocumentor

Nevýhodou PhpDocumentor je nepodpora[[78]](#footnote-78) kódování UTF-8. Nicméně je možné ji doplnit úpravou[[79]](#footnote-79) šablon pro generování dokumentace.

### Docblox

Docblox[[80]](#footnote-80) je nový nástroj na generování API dokumentace. Jednou z jeho předností je rychlost generování. To je dobře vidět například na dokumentaci Zend Frameworku, která se s využitím nástroje PhpDocumentor generovala 100 minut, tak DocBlox ji zvládne vygenerovat za přibližně 10 minut[[81]](#footnote-81). Další zajímavou funkcí je generování schématu tříd ve formátu SVG.

Generování dokumentace v DocBloxu se spouští velmi podobně jako v případě nástroje PhpDocumentor:

docblox -f phpdoc01.php -t docblox

Jak vypadá vygenerovaná dokumentace ukazuje Obrázek 7.



Obrázek 7: Ukázka dokumentace vygenerované nástrojem DocBlox

#### Validace dokumentačních komentářů

Docblox při generování dokumentace zároveň kontroluje, zda dokumentační komentáře u zdrojového kódu obsahují potřebné informace. Případné nedostatky standardně reportuje do sekce v HTML dokumentaci. Pro snadnější napojení na kontinuální integraci existuje šablona „checkstyle“[[82]](#footnote-82), která přehled problematických míst vygeneruje do XML souboru.

Šablonu je možné nainstalovat přímo přes Docblox:

docblox template:install checkstyle

Upravený příkaz pro kontrolu dokumentace vypadá takto:

docblox -f phpdoc01.php -t docblox --template checkstyle

### ApiGen

ApiGen[[83]](#footnote-83) je další nový nástroj na generování dokumentace ze zdrojových kódů v jazyce PHP. Je zajímavý například tím, že podporoval generování dokumentace pro PHP 5.4 už dříve, než bylo vydáno, a také tím, že je vyvíjen v České republice.

Instalaci je možné snadno provést pomocí nástroje PEAR:

pear config-set auto\_discover 1

pear install pear.apigen.org/apigen

V případě problémů s instalací pomocí PEAR je možné si stáhnout[[84]](#footnote-84) instalační balíček, který již obsahuje všechny potřebné knihovny. Stačí ho jen rozbalit a jeho umístění přidat do systémové cesty.

Spustit ho lze opět snadno pomocí příkazové řádky:

apigen -s phpdoc01.php -d apigen

Jak vypadá vygenerovaná dokumentace ukazuje Obrázek 8.



Obrázek 8: Ukázka dokumentace vygenerovaná nástrojem ApiGen

#### Validace dokumentačních komentářů

ApiGen stejně jako DocBlox provádí kontrolu, zda jsou dokumentační komentáře vyplněny a případné chyby také umožňuje exportovat do formátu pro Checkstyle:

apigen -s phpdoc01.php -d apigen --report apigen/checkstyle.xml

### Výběr nástroje pro generování dokumentace

Pro volbě nástroje pro generování dokumentace ze zdrojových kódů máme na výběr z několika možností. Vzhledem k tomu, že ostatní součásti kontinuální integrace na vygenerované dokumentaci nezávisí, je možné nástroj snadno vyměnit i později.

Nakonec jsem pro využití v kontinuální integraci zvolil nástroj ApiGen, protože má subjektivně přehlednější dokumentaci než DocBlox. PhpDocumentor jsem nevybral, protože neumožňuje vytvoření reportu pro Checkstyle.

## Automatizace sestavení

Kontinuální integrace vyžaduje pro své fungování automatizované sestavení (viz kapitola 3.1.2) Integrační server Jenkins, zvolený pro implementaci (viz kapitola 4.2), podporuje pomocí rozšíření různé nástroje na automatizaci sestavení. Například Apache Ant[[85]](#footnote-85), Phing[[86]](#footnote-86), Rake[[87]](#footnote-87), Maven[[88]](#footnote-88), NAnt[[89]](#footnote-89) a další.

Pro tuto implementaci integračního serveru jsem zvažoval nástroje Apache Ant a Phing. Ant je nástroj typicky využívaný pro sestavování projektů napsaných v jazyce Java. Phing vznikl jako převod principů využívaných v nástroji Ant do jazyka PHP.

Oba nástroje využívají k popisu procesu sestavení XML soubory, které obsahují jednotlivé úkoly. Kořenovým elementem dokumentu je tzv. projekt (project), který obaluje jednotlivé úkoly (target). Úkoly se skládají z jednotlivých úloh (task).

Ukázkový soubor může vypadat takto:

//build-example.xml

<?xml version="1.0" encoding="UTF-8"?>

<project name="Ukazkove sestaveni" default="build">

<target name="cleanup">

<echo message="Zde by byly dalsi ulohy pro vycisteni pracovniho adresare"></echo>

</target>

<target name="build" depends="cleanup">

<echo message="Zde by byly dalsi ulohy pro sestaveni"></echo>

</target>

</project>

Projekt má v atributu default určený výchozí úkol, který se provede, pokud nebude při spuštění určeno jinak. U úkolu build je uvedená závislost na úkolu cleanup, což znamená, že úkol build se neprovede dříve, než bude vyčištěn pracovní adresář.

Detailní popis nástrojů Ant a Phing přesahuje rozsah této práce, více informací lze nalézt v [THE PHING PROJECT, 2011], [ZANDSTRA, 2007] nebo [LOUGHRAN, 2007].

Tento jednoduchý skript pro sestavení využívá jen vlastnosti společné jak pro Ant, tak pro Phing, takže je možné ho spustit pomocí obou:

Phing:

> phing -f build-example.xml

Buildfile: I:\\_BP\phing\build-example.xml

Ukazkove sestaveni > cleanup:

[echo] Zde by byly dalsi ulohy pro vycisteni pracovniho adresare

Ukazkove sestaveni > build:

[echo] Zde by byly dalsi ulohy pro sestaveni

BUILD FINISHED

Total time: 0.2688 seconds

Ant:

> ant.bat -f build-example.xml

Buildfile: I:\\_BP\phing\build-example.xml

cleanup:

[echo] Zde by byly dalsi ulohy pro vycisteni pracovniho adresare

build:

[echo] Zde by byly dalsi ulohy pro sestaveni

BUILD SUCCESSFUL

Total time: 0 seconds

Jak je vidět, tak výstupy se u takto jednoduchého skriptu liší jen v drobných detailech.

### Výběr vhodného nástroje pro automatizaci sestavení

Pro účely nasazení kontinuální integrace jsem zvolil nástroj Phing a to z několika důvodů:

Je implementovaný v jazyce PHP a je zaměřený na automatizaci úloh, které mohou být při jeho využití potřebné. Pro většinu z nich už obsahuje vytvořené šablon úloh, takže jejich parametry lze zapisovat přímo jako elementy a atributy v XML a ne jen jako parametry příkazové řádky. Díky tomu, že celá implementace je v PHP, tak je možné si snadno dopsat vlastní šablony úloh.

V neposlední řadě jsem ho zvolil proto, že už se osvědčil při automatizaci nasazování PHP aplikací ve společnosti w3w, s.r.o.

Instalace nástroje Phing je snadná, lze ji provést pomocí již dříve popsaného nástroje PEAR (viz kapitola 5.1):

pear channel-discover pear.phing.info

pear install phing/phing

## HipHop

hiphop analysis - měl by dělat i validaci PHP, lepší než samotné php :)

http://sebastian-bergmann.de/archives/894-Using-HipHop-for-Static-Analysis.html

http://sebastian-bergmann.de/archives/918-Static-Analysis-with-HipHop-for-PHP.html

https://github.com/sebastianbergmann/hphpa

## 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

CI crawl & check for 200 - místo testování, jen zběžná kontrola, jestli se něco nerozbilo

# Praktická implementace CI platformy v malé firmě

V této kapitole se zaměřím na postup instalace CI platformy pro malou firmu, nebudu se tedy zabývat tématy, která se netýkají kontinuální integrace v malé firmě, jako je koordinace spolupráce více integračních serverů, distribuované spouštění testů a další.

## Skript pro sestavení aplikace

Vývoj a testování skriptu pro sestavení je vhodné provádět spíše na menším projektu než na rozsáhlé aplikaci. Důvodem je, že při volbě rozsáhlé aplikace se stovkami PHP souborů, by bylo nutné zbytečně dlouhou dobu čekat na proběhnutí jednotlivých částí sestavení.

Proto jsem pro vytvoření skriptu pro sestavení aplikace zvolil ukázkovou aplikaci ZFTutorial, která je výsledkem návodu Getting Started with Zend Framework[[90]](#footnote-90), pro úvod do funkčnosti Zend Frameworku[[91]](#footnote-91).

Protože aplikace je k dispozici jen jako archív ve formátu ZIP a pro kontinuální integraci je důležité využívat verzovací systém, vytvořil jsem GIT repositář[[92]](#footnote-92) na serveru GitHub.com a aplikaci tam uložil.

@todo Shopio budu integrovat jen interně a budu z něj dělat ukázky metrik

Jako první krok tvorby skriptu pro sestavení aplikace je nutné si v kořenovém adresáři projektu vytvořit prázdný soubor build.xml. Do něj vložíme kostru projektu a jednoduchý „task“ echo, abychom ověřili, že Phing funguje správně.

<project name="zf-tutorial" default="main">

<target name="main">

<echo message="Phing works!"/>

</target>

</project>

Po zavolání příkazu phing v adresáři, kde je uložen soubor build.xml bychom měli dostat tento výstup:

> phing

Buildfile: I:\\_BP\zf-tutorial\build.xml

zf-tutorial > main:

[echo] Phing works!

BUILD FINISHED

Total time: 0.3852 seconds

## PHP Lint

Prvním krokem, který je nutné při sestavení udělat je kontrola syntaktické správnosti, pro jazyk PHP je to tzv. PHP Lint (viz kapitola 5.2).

Do build skriptu přidáme nový target:

<target name="lint">

<phplint haltonfailure="true">

<fileset dir="${project.basedir}/application">

<include name="\*\*/\*.php"/>

<include name="\*\*/\*.phtml"/>

</fileset>

</phplint>

</target>

Je zde několik věcí, které je potřeba vysvětlit. Phing již v základu obsahuje task phplint[[93]](#footnote-93), který na pozadí spouští samotnou kontrolu pomocí php -l.

Atribut haltonfailure="true" znamená, že pokud selže syntaktická kontrola jakéhokoliv souboru, tak ihned selže celý build a nebudou se provádět další kroky (například analýza coding standards).

Proměnná ${project.basedir} se automaticky nastaví na adresář, který je nadřazený souboru build.xml.

Sada souborů (fileset) popisuje množinu souborů, která se předají úloze (task). Bylo by možné stejný fileset zkopírovat do dalších částí skriptu pro sestavení, protože na zdrojových kódech samotné aplikace se spouští i další úlohy. Nicméně Phing toto umožňuje řešit elegantněji. Je možné vytvořit fileset i uvnitř tagu <project>, přiřadit mi identifikátor a na něj pak jen odkazovat.

Fileset si označíme jako „src“:

<fileset id="src" dir="${project.basedir}/application">

<include name="\*\*/\*.php"/>

<include name="\*\*/\*.phtml"/>

</fileset>

A target pak můžeme zpřehlednit takto:

<target name="lint">

<phplint haltonfailure="true">

<fileset refid="src"/>

</phplint>

</target>

Podobně lze ve skriptu pro sestavení nadefinovat adresáře tests a library a ty doplnit do úkolu phplint.

Jak vypadá výsledný skript, je možné vidět v {@todo příloha 1}.

## Vytvoření projektu na integračním serveru

Integrační server je nainstalovaný a nastavený podle postupu v kapitolách 4.3 a 4.4. Vše je tedy připravené a je možné se pustit do přípravy automatizovaného sestavování. Je nutné nainstalovat dvě rozšíření - GIT[[94]](#footnote-94) a Phing[[95]](#footnote-95).

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin phing

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin git

Instalace verzovacího systému GIT přesahuje rozsah této práce, pro další postup předpokládáme, že je k dispozici v systémové cestě. Více informací lze nalézt například v [CHACON, 2009].

Na úvodní stránce zvolíme možnost „New Job“ („Job“ je označení, které Jenkins používá pro něco, co chápeme jako „projekt“), vyplníme jméno (v našem případě zf-tutorial), dále vybereme volbu „Build a free-style software project“. @todo doplnit obrázky

V dalším kroku v sekci „Source Code Management“ vložíme URL repositáře s naším projektem (https://mhujer@github.com/mhujer/zf-tutorial.git) a do pole „Branches to build“ zadáme „master“ (chceme sestavení provádět jen na hlavní větvi projektu).

V sekci „Build“ zvolíme „Add build step“ a „Invoke Phing targets“. Nic dalšího není nutné nastavovat, protože soubor s instrukcemi pro sestavení je pojmenovaný standardně, a zároveň je nastavený výchozí „target“. Stačí tedy změny uložit pomocí volby „Save“ a poté spustit sestavení pomocí „Build Now“ v nabídce vlevo.

Vidíme, že sestavení proběhlo v pořádku.



Obrázek 9: Úspěšné sestavení v Jenkinsu

Do detailních informací o sestavení se dostaneme kliknutím na datum a čas konkrétního sestavení v části „Build History“. V sekci „Console Output“ vidíme, průběh sestavení (pokud by build trval déle, tak tu lze vidět aktuální průběh).

Dále je vidět, že se spustil target main, který jen vypsal informační hlášku.

zf-tutorial > main:

[echo] Phing works!

Přejdeme proto do nastavení projektu a sekci „Build“ vyplníme do pole „Targets“ „lint“, což je název targetu, který jsme si vytvořili v kapitole 6.2. Spustíme další sestavení pomocí „Build Now“.

V konzoli teď vidíme, že už se build spustil tak, jak měl:

zf-tutorial > lint:

[phplint] I:\BP-jenkins\data\jobs\zf-tutorial\workspace\application\Bootstrap.php: No syntax errors detected

.........

[phplint] I:\BP-jenkins\data\jobs\zf-tutorial\workspace\tests\library\bootstrap.php: No syntax errors detected

BUILD FINISHED

Total time: 1.0731 second

Finished: SUCCESS

## Nasazení PHPUnit

Nástroj pro automatizované testování je popsán v kapitole 5.8. Do skriptu pro sestavení aplikace ho lze přidat pomocí:

<target name="phpunit" depends="prepare" description="PHPUnit testy">

<phpunit printsummary="true" haltonfailure="true" haltonerror="true">

<formatter todir="${project.basedir}/build" outfile="phpunit-report.xml" type="xml"/>

<batchtest>

<fileset refid="tests"/>

</batchtest>

</phpunit>

</target>

Je vhodné ho zařadit ihned po kontrole pomocí PHPLint, protože opět platí, že pokud automatizované testy odhalí chyby ve funkčnosti, tak další metriky kódu nemá smysl získávat. Důležité jsou pro to atributy haltonfailure a haltonerror, které v případě selhání testu nebo chyby ve skriptu zastaví zpracování sestavení.

Pro Jenkins existuje rozšíření, které slouží ke zpracování výsledků testů ze všech nástrojů z rodiny xUnit[[96]](#footnote-96). Nainstalovat ho lze pomocí příkazu:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin xunit

Po restartu serveru je nutné rozšíření nakonfigurovat ve vlastnostech projektu. Prvním krokem je zaškrtnutí pole „Publish testing tools result report“, dalším přidání „PHPUnit-3.4“ pomocí tlačítka „Add“. Do pole „Pattern“ je nutné vyplnit relativní cestu k výsledku proběhnutí testů - build/phpunit-report.xml. Po sestavení projektu na úvodní stránce přibude položka „Latest Test Result“, kde je možné procházet výsledky testů.

### Pokrytí kódu testy

Zjišťování metriky pokrytí kódu testy jsem se rozhodl nenasazovat z důvodů popsaných v kapitole 3.3.3.1. Více informací o zařazení této metriky do kontinuální integrace lze nalézt v [BERGMANN, 2011b]

## Nasazení PHP\_CodeSniffer

Do skriptu pro sestavení jsem přidal kontrolu Coding Standards s použitím standardu pro Zend Framework a exportem chyb ve formátu pro Checkstyle.

<target name="phpcs" depends="prepare" description="Kontrola Coding standards">

<phpcodesniffer standard="Zend">

<fileset refid="src"/>

<fileset refid="tests"/>

<fileset refid="library"/>

<formatter type="default" usefile="false"/>

<formatter type="checkstyle" outfile="${project.basedir}/build/checkstyle-phpcs.xml"/>

</phpcodesniffer>

</target>

Dále jsem přidal další target, v tomto případě nebude vykonávat žádnou činnost, ale bude záviset na ostatních, takže zajistí, že ty se spustí dříve.

<target name="build" depends="prepare, lint, phpcs" description="Meta target, spouští ostatní targety"/>

Vzhledem k tomu, že PHP\_CodeSniffer generuje tzv. artefakt (soubor, který je výsledkem buildu), je nutné při dalších spuštěních zajistit, aby tento soubor neexistoval a vygeneroval se znovu. Toho lze dosáhnout pomocí dalších targetů, jeden pro smazání adresáře a druhý pro jeho znovuvytvoření. Je vhodné upravit všechny ostatní targety, aby závisely na targetu prepare, neboť je pak bude možné spouštět i samostatně.

<target name="cleanup" description="Vyčistění workspace">

<delete dir="${project.basedir}/build"/>

</target>

<target name="prepare" depends="cleanup" description="Příprava workspace">

<mkdir dir="${project.basedir}/build"/>

</target>

Zároveň bylo nutné upravit zdrojový kód ukázkové aplikace, aby tam existovalo nějaké porušení coding standards.

Do Jenkinse je nutné doinstalovat rozšíření Checkstyle, které umí zpracovat XML soubor se zjištěnými chybami:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin checkstyle

Po restartu serveru je dále nutné v editaci projektu v sekci Post-build Actions vybrat volbu Publish Checkstyle analysis results a do pole Checkstyle results vyplnit build/checkstyle-phpcs.xml. Jenkins vypíše chybové hlášení, že soubor neexistuje, což je správně, protože bude vygenerován až při dalším sestavení.

Posledním krokem je odstranění názvu vybraného targetu v sekci Build (použije se tedy výchozí nastavený v skriptu pro sestavení).

## Nasazení ApiGen

Dalším krokem je přidání generování dokumentace pomocí nástroje ApiGen popsaného v kapitole 5.9.3.

Nejprve je nutné přidat target pro ApiGen (je využit přímo task ApiGen, který už je součástí testovací verze nástroje Phing, nicméně není součástí existující stabilní verze 2.4.9):

<target name="apigen" depends="prepare" description="Generování dokumentace">

<apigen source="${project.basedir}/application" destination="${project.basedir}/build/docs" report="${project.basedir}/build/checkstyle-apigen.xml"/>

</target>

Jenkins je schopen na hlavní stránku projektu doplnit odkazy na další HTML výstupy, toho lze dosáhnout rozšířením HTML Publisher[[97]](#footnote-97), které nainstalujeme pomocí:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin htmlpublisher

Po restartu je třeba HTML Publisher nakonfigurovat. V editaci nastavení projektu v sekci Post-build Actions musí být zaškrtnuté pole Publish HTML reports. Adresář s daty (HTML directory to archive) je build/docs. Report title je vhodné nastavit například na „API Docs“. Ostatní volby je možné ponechat ve výchozím nastavení.

ApiGen generuje i soubor s chybami v dokumentačních komentářích, pročež je nutné upravit pole „Checkstyle results“ na build/checkstyle-\*.xml (hvězdičkou dosáhneme toho, že se budou načítat všechny XML soubory z daného adresáře začínající na checkstyle-.

## Nasazení PHPCPD

Nástroj na kontrolu duplicitního kódu PHPCPD je detailně popsán v kapitole 5.4. Je nutné ho zařadit do skriptu pro sestavení. Opět je možné využít již předpřipravený task v rámci nástroje Phing:

<target name="phpcpd" depends="prepare">

<phpcpd>

<fileset refid="src"/>

<fileset refid="tests"/>

<fileset refid="library"/>

<formatter type="pmd" outfile="${project.basedir}/build/pmd-cpd.xml"/>

</phpcpd>

</target>

Aby bylo možné ověřit, že nástroj kontroluje kód správně, je vhodné například zduplikovat nějaký PHP soubor v testovací aplikaci (bude nutné přejmenovat třídu v něm, aby nedošlo k problémům při generování dokumentace).

Pro Jenkins existuje přímo rozšíření, které umí zpracovat výsledky analýzy duplicitního kódu, DRY (zkratka pro Don't Repeat Yourself). Instalaci lze provést pomocí příkazu:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin dry

Zároveň je nutné nastavit projekt v Jenkinsu. Zaškrtnutím pole Publish duplicate code analysis results v sekci Post-build Actions se aktivují další pole. Do Duplicate code results vložíme cestu k souboru, který nám PHPCPD generuje: build/pmd-cpd.xml.

Po provedení sestavení máme k dispozici i informace o duplicitním kódu.

## Nasazení PHPLOC

PHPLOC je nástroj, který počítá různé metriky zdrojových kódů a výsledky umí exportovat do CSV souboru využitelného v Jenkinsu. Detailně byl popsán v kapitole 5.5.

Do skriptu pro sestavení se spouštění PHPLOC doplní přidáním další targetu:

<target name="phploc" depends="prepare" description="Analýza PHPLOC">

<exec command="phploc --log-csv ${project.basedir}/build/phploc.csv ${project.basedir}/application" logoutput="true" />

</target>

Do Jenkinse je nutné nainstalovat rozšíření Plot[[98]](#footnote-98), které umí vykreslovat grafy z hodnot získaných v jednotlivých sestaveních.

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin plot

@todo Popis jeho detailního nastavení přesahuje rozsah této práce (a nebo to radši rozepíšu?).

Nastavení je proto možné buď vytvořit ručně podle tabulek;

XML: je připraveno v samostatném XML souboru https://github.com/mhujer/bakalarka/blob/master/jenkins/plot.xml. Ten do Jenkinse n@todo

## Nasazení PDepend

PDepend je nástroj umožňující získávat různé metriky zdrojových kódů. Jeho detailní popis je možné nalézt v kapitole 5.5.

Do skriptu pro sestavení ho lze přidat opět pomocí předpřipraveného tasku v nástroji Phing. Dostal opět jen soubory, které jsou přímo součástí aplikace, nebudeme metriky získávat pro použité knihovny apod. PDepend generuje jak data pro rozšíření JDepend, tak i dvě přehledová schémata.

<target name="pdepend" depends="prepare" description="Analýza nástrojem PDepend">

<phpdepend>

<fileset refid="src"/>

<logger type="jdepend-xml" outfile="${project.basedir}/build/jdepend.xml"/>

<logger type="jdepend-chart" outfile="${project.basedir}/build/dependencies.svg"/>

<logger type="overview-pyramid" outfile="${project.basedir}/build/overview-pyramid.svg"/>

</phpdepend>

</target>

Do Jenkinse je nutné nainstalovat rozšíření JDepend[[99]](#footnote-99):

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin jdepend

Po restartu serveru je nutné mu nastavit cestu k souboru, který se generuje během sestavení. V sekci Post-build Actions je nutné zaškrtnout pole Report JDepend a do pole Pre-generated JDepend File zadat cestu: build/jdepend.xml.

Na zobrazení vygenerovaných schémat není nutné do Jenkinse instalovat žádné rozšíření. Vzhledem k tomu, že jsou to obrázky ve formátu SVG, které umí dnešní běžně využívané prohlížeče bez problémů zobrazit, tak stačí na hlavní stránce projektu kliknout na odkaz „add description“ a do textového pole vložit tento kód:

<img src="http://localhost:8080/job/zf-tutorial/ws/build/dependencies.svg"/>

<img src="http://localhost:8080/job/zf-tutorial/ws/build/overview-pyramid.svg"/>

Schémata se poté budou zobrazovat na hlavní stránce.

## Nasazení PHPMD

PHPMD (Mess Detector) je nástroj na odhalení potenciálních problémů ve zdrojových kódech. Jeho detailnější popis lze nalézt v kapitole 5.7.

Skript pro sestavení je nutné doplnit o následující kód:

<target name="phpmd" depends="prepare" description="PMD analýza">

<phpmd rulesets="${project.basedir}/phpmd.xml">

<fileset refid="src"/>

<formatter type="xml" outfile="${project.basedir}/build/pmd.xml"/>

</phpmd>

</target>

Podporu pro zpracování výstupů z PHPMD lze do Jenkinse doplnit pomocí příkazu:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin pmd

Po restartu je nutné nastavit načítání výstupů v našem projektu. V sekci „Publish PMD analysis results“ vložíme do pole „PMD results“ cestu k souboru build/pmd.xml. Výsledky analýzy je poté možné procházet v sekci „PMD Warnings“.

## Nasazení rozšíření Violations

Jednotlivé nástroje zařazené do skriptu pro sestavení nám generují reporty, které pak jednotlivá rozšíření Jenkinse zpracovávají. Bylo by vhodné mít ucelený pohled na problémy ve zdrojových kódech. Pro Jenkins existuje rozšíření Violations[[100]](#footnote-100), které tento problém řeší.

Nejprve je nutné ho nainstalovat:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin violations

Vytváření shrnujících reportů lze aktivovat v nastavení projektu - zaškrtneme pole „Report Violations“ a nově zobrazená pole vyplníme takto:

checkstyle: build/checkstyle-\*xml

cpd: build/pmd-cpd.xml

pmd: build/pmd.xml

## Nasazení PHP\_CodeBrowser

Druhým nástrojem, který umí generovat shrnující reporty problémů v kódu je PHP\_CodeBrowser[[101]](#footnote-101). Je vyvinutý přímo pro zpracovávání výstupů z analýz zdrojových kódů v jazyce PHP. Výsledkem jeho činnosti je sada HTML souborů, kde je procházitelný strom souborů a v nich jsou vyznačena problematická místa.

Nainstalujeme ho pomocí nástroje PEAR:

pear install phpunit/PHP\_CodeBrowser

Do skriptu pro sestavení ho přidáme takto:

<target name="phpcb" depends="phpcs, phpcpd, phpmd" description="Vygeneruje souhrnný výstup chyb v kódu pomocí PHP\_CodeBrowser">

<exec command="phpcb --log ${project.basedir}/build --source ${project.basedir} --output ${project.basedir}/build/code-browser" logoutput="true" />

</target>

A podobně jako v případě vygenerované dokumentace v AgiGenu (viz kapitola 6.6) ho musíme doplnit do nastavení rozšíření HTML Publisher. V nastavení projektu v sekci „Post-build Actions“ v části „Publish HTML reports“ vytvoříme nový řádek. Do pole „HTML directory to archive“ vložíme build/code-browser/ a titulek („Report title“) nastavíme například na „Code Browser“.

Není nutné instalovat oba nástroje - PHP\_CodeBrowser a Violations, záleží na tom, který se nám subjektivně bude zdát vhodnější pro naši instalaci serveru.

## Finální nastavení

SCM polling

## Úprava firemních procesů

* praktické použití na nějakém projektu
* //integrační problémy při použití branchí
* - "artefakty"
* - -

# Závěr

@todo

shrnutí výsledků, ke kterým autor dospěl,

přínos autora práce k řešené problematice (co je v práci původní),

zhodnocení využitelnosti dosažených výsledků, příp. další náměty pro řešení v uvedené

oblasti.

Můj přínos

* Phing místo Antu jako ostatní
* validace dokumentace
* ApiGen bugreporty
* Nové tasky do Phingu? @todo PHP\_CodeBrowser, PHPLoc Phing tasky + upravit RNG schéma
* …

https://github.com/apigen/apigen/issues/113

https://github.com/apigen/apigen/issues/114

https://github.com/apigen/apigen/issues/119

https://github.com/phingofficial/phing/pull/100

<https://github.com/phingofficial/phing/pull/105>

<https://github.com/pdepend/pdepend/issues/87>

<https://github.com/zendframework/zf2/pull/992>

<https://github.com/zendframework/zf2/pull/993>

* Má Zend Framework Continous Integration server?
* <http://www.youbrokethebuild.com/>
  + <http://www.papercut.com/blog/chris/2011/08/19/who-broke-the-build/>
* Continuous Integration anti-patterns <http://www.ibm.com/developerworks/java/library/j-ap11297/>¨

Automatizace automatizace

* php project wizard https://github.com/sebastianbergmann/php-project-wizard

Celé by to šlo vzít z https://github.com/sebastianbergmann/php-jenkins-template/blob/master/config.xml ale je naprd mít blackbox, který nechápeme.

<http://reload.github.com/phing-drupal-template/>

Zbuildovat Shopio po revizích – alespoň 100

Poll SCM – proč v průběhu práce ručně?

<https://github.com/abtris/vagrant-hudson>

Překlad Jenkinse? Ať si ustanovím terminologii ☺

Bezpečnostní testování <https://www.fortify.com/ssa-elements/threat-intelligence/rats.html>

Zkontrolovat proti <http://kit.vse.cz/bakalarske-studium/bakalarske-prace/>

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

# Terminologický slovník @todo

|  |  |  |
| --- | --- | --- |
| commit |  |  |
| build | sestavení |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Seznam použité literatury a zdrojů

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

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

BERGMANN, S. *PHPUnit Pocket Guide*. O'Reilly Media, 2005. ISBN 978-0596101039.

BERGMANN, S. Benchmark of PHP Branches 3.0 through 5.3-CVS [online]. 07. 02. 2008 [cit. 2011-07-20]. Dostupné z: http:/​/​sebastian-bergmann.de/​archives/​745-Benchmark-of-PHP-Branches-3.0-through-5.3-CVS.html

BERGMANN, S. a S. PRIEBSCH. *Real-World Solutions for Developing High-Quality PHP Frameworks and Applications*. Indianapolis: Wrox, 2011. ISBN 978-0470872499.

BERGMANN, S. *Integrating PHP Projects with Jenkins*. Sebastopol: O'Reilly Media, Inc., 2011b. ISBN 978-1-4493-0943-5.

CRAG, R. D. a S. P. JASKIEL. *Systematic software testing*. Artech House, 2002. ISBN 1580535089.

DEMARCO, T. *Controlling software projects: management, measurement & estimation*. Michigan: Yourdon Press, 1982. ISBN 9780917072321.

DUVALL, P., S. MATYAS a A. GLOVER. *Continuous Integration: Improving Software Quality and Reducing Risk*. Upper Saddle River, NJ: Addison-Wesley Professional, 2007. ISBN: 978-0321336385.

FOWLER, M. *Continuous Integration* [online]. 1. 5. 2006 [cit. 2011-07-18]. Dostupné z: http:/​/​martinfowler.com/​articles/​continuousIntegration.html

HUMBLE, J. a D. FARLEY. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN: 9780321601919.

CHACON, S. *Pro Git*. Apress, 2009. ISBN 978-1430218333.

KOSKELA, L. JavaRanch Journal *Introduction to Code Coverage* [online]. © 2004 [cit. 2012-03-31]. Dostupné z: http:/​/​www.javaranch.com/​journal/​2004/​01/​IntroToCodeCoverage.html

LOUGHRAN, S. a H. ERIK. *Ant in Action*. Greenwich: Manning, 2007. ISBN 1-932394-80-X.

MARTIN, R. C. Object Mentor *Coding Standards* [online]. 18. 4. 2007 [cit. 2012-03-17]. Dostupné z: http:/​/​blog.objectmentor.com/​articles/​2007/​04/​18/​coding-standards

MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Boston: Prentice Hall, 2008. ISBN: 978-0132350884.

MCCABE, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 1976,(4), 308-320. ISSN 0098-5589.

MCCONNELL, S. *Odhadování softwarových projektů*. Brno: Computer Press, 2006. ISBN 80-251-1240-3.

MCCONNELL, S. 10x Software Development *Measuring Productivity of Individual Programmers* [online]. © 2008 [cit. 2012-04-01]. Dostupné z: http:/​/​forums.construx.com/​blogs/​stevemcc/​archive/​2008/​04/​09/​measuring-productivity-of-individual-programmers.aspx

MENŠÍK, V. ET NETERA *CI za použití Hudsonu a Mavenu* [online]. 20. 10. 2009 [cit. 2011-07-27]. Dostupné z: http:/​/​www.etnetera.cz/​cz/​21447-tech\_streda/​ci\_za\_pouziti\_hudsonu\_a\_mavenu.html

MYERS, G. J. *The Art of Software Testing, Second Edition*. Wiley, 2004. ISBN 978-0471469124.

MYTTON, D. SitePoint *Why You Need Coding Standards* [online]. 23. 9. 2004 [cit. 2012-03-17]. Dostupné z: http:/​/​www.sitepoint.com/​coding-standards/

NEJMEH, B. A. NPATH: a measure of execution path complexity and its applications. *Communications of the ACM*. New York, NY, USA: ACM, 1988,(31), 188--200. ISSN 0001-0782.

THE PHING PROJECT *Phing 2.4 - User Guide* [online]. © 2011, verze 1b3461f9f8e5da4becf97e7c298e209f1e6c8c6c [cit. 2011-03-29]. Dostupné z: http:/​/​www.phing.info/​docs/​guide/​stable/

PICHLER, M. Cyclomatic Complexity *PHP Depend* [online]. © 2011, verze 24.1.2012 [cit. 2012-04-02]. Dostupné z: http:/​/​pdepend.org/​documentation/​software-metrics/​cyclomatic-complexity.html

SMART, J. F. *Jenkins: The Definitive Guide*. Sebastopol: O'Reilly Media, 2011. ISBN 978-1-4493-8959-8.

VAN DAM, M. Zend.com *Improving QA on PHP development projects* [online]. 14. 4. 2011 [cit. 2012-02-29]. Dostupné z: http:/​/​www.zend.com/​en/​resources/​webinars/​#QAPHP

WOOD, J. *Why Code Coverage Alone Doesn’t Mean Squat* [online]. © 2008 [cit. 2012-03-31]. Dostupné z: http:/​/​johnpwood.net/​2008/​12/​30/​why-code-coverage-alone-doesnt-mean-squat/

WIKIPEDIA CONTRIBUTORS. Wikipedia, The Free Encyclopedia *Continuous integration* [online]. 20. 07. 2011 [cit. 2011-07-27]. Dostupné z: http:/​/​en.wikipedia.org/​w/​index.php?title=Continuous\_integration&oldid=440518476

ZANDSTRA, M. *PHP Objects, Patterns, and Practice*. Apress, 2007. ISBN 978-1590599099.

ZIKMUND, Š. *Diplomová práce: Deployment aplikací v PHP*. Praha: KIT FIS VŠE, 2011

# Seznam obrázků, tabulek a ukázek kódu

## Seznam obrázků

[Obrázek 1: Schéma vodopádového modelu vývoje software 8](#_Toc321126634)

[Obrázek 2: "V" model testování software [CRAG, 2002] 15](#_Toc321126635)

[Obrázek 3: Hlavní stránka Jenkinsu po prvním spuštění 23](#_Toc321126636)

[Obrázek 4: Jenkins - nastavení anglického prostředí 24](#_Toc321126637)

[Obrázek 5: Kontrola syntaktických chyb v editoru Zend Studio 8 26](#_Toc321126638)

[Obrázek 6: Ukázka dokumentace vygenerované pomocí skriptu PhpDocumentor 33](#_Toc321126639)

[Obrázek 7: Ukázka dokumentace vygenerované nástrojem DocBlox 34](#_Toc321126640)

[Obrázek 8: Ukázka dokumentace vygenerovaná nástrojem ApiGen 36](#_Toc321126641)

[Obrázek 9: Úspěšné sestavení v Jenkinsu 42](#_Toc321126642)

## Seznam tabulek

[Tabulka 1: Srovnání aktivity u projektů Hudson a Jenkins na serveru GitHub.com 21](#_Toc321126643)

## Seznam ukázek kódu

[Kód 3.1 Míra pokrytí kódu testy 16](#_Toc321126644)

[Kód 5.1 test1-error.php 26](#_Toc321126645)

[Kód 5.2 Chybový výstup z PHP Lint 26](#_Toc321126646)

[Kód 5.3 Zdrojový kód souboru test2-ok.php 26](#_Toc321126647)

[Kód 5.4 Výstup z PHP Lint, když je kód v pořádku 26](#_Toc321126648)

[Kód 5.5 test3-trait.php 27](#_Toc321126649)

[Kód 5.6 Výstup z PHP Lint (verze PHP 5.3) na souboru test3-trait.php 27](#_Toc321126650)

[Kód 5.7 Chybový výstup z PHP Lint (verze PHP 5.4.0alpha2) na souboru test3-trait.php 27](#_Toc321126651)

[Kód 5.8 Příklad šablony v PHTML souboru (test4-template.phtml) 28](#_Toc321126652)

[Kód 5.9 Ukázka použití PHP CPD 29](#_Toc321126653)

[Kód 5.10 Ukázkový výstup z aplikace phploc 30](#_Toc321126654)

[Kód 5.11: Ukázka možnosti využití PHPDoc (soubor phpdoc01.php) 32](#_Toc321126655)

# Rejstřík

# Přílohy

1. QA = quality assurance – kontrola kvality [↑](#footnote-ref-1)
2. Volně podle [WIKIPEDIA CONTRIBUTORS, 2011] [↑](#footnote-ref-2)
3. Přeloženo z [FOWLER, 2006] [↑](#footnote-ref-3)
4. Viz http://www.extremeprogramming.org/rules/integrateoften.html [↑](#footnote-ref-4)
5. Zkracováno jako EP - viz [BECK, 2005] [↑](#footnote-ref-5)
6. SCM - Source Code Management Tools [↑](#footnote-ref-6)
7. Viz http://subversion.tigris.org/ [↑](#footnote-ref-7)
8. Concurrent Versions System – viz http://savannah.nongnu.org/projects/cvs [↑](#footnote-ref-8)
9. Viz https://jazz.net/projects/rational-team-concert/ [↑](#footnote-ref-9)
10. Viz http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview [↑](#footnote-ref-10)
11. Viz http://git-scm.com/ [↑](#footnote-ref-11)
12. Viz http://mercurial.selenic.com/ [↑](#footnote-ref-12)
13. Viz http://bazaar.canonical.com/ [↑](#footnote-ref-13)
14. Viz http://www.bitkeeper.com/ [↑](#footnote-ref-14)
15. Viz http://kerneltrap.org/node/4966 [↑](#footnote-ref-15)
16. Získání kopie všech souborů z repositáře [↑](#footnote-ref-16)
17. Tzv. build [↑](#footnote-ref-17)
18. Java Development Kit http://www.oracle.com/technetwork/java/javase/downloads/index.html [↑](#footnote-ref-18)
19. Integrated development environment neboli Vývojové prostředí [↑](#footnote-ref-19)
20. Viz http://www.gnu.org/software/make/ [↑](#footnote-ref-20)
21. Viz http://ant.apache.org/ [↑](#footnote-ref-21)
22. Viz http://www.phing.info [↑](#footnote-ref-22)
23. TDD neboli vývoj řízený testy [BECK, 2002] [↑](#footnote-ref-23)
24. Viz http://www.martinfowler.com/bliki/Xunit.html [↑](#footnote-ref-24)
25. Viz http://www.junit.org/ [↑](#footnote-ref-25)
26. Viz http://www.nunit.org/ [↑](#footnote-ref-26)
27. Viz http://www.phpunit.de/ [↑](#footnote-ref-27)
28. 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-28)
29. Viz kapitola 3.3 [↑](#footnote-ref-29)
30. Viz http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBuildsInTrouble.rdoc [↑](#footnote-ref-30)
31. http://dattein.com/blog/arduino-build-light/ [↑](#footnote-ref-31)
32. http://fabiopereira.me/blog/2009/12/15/build-dashboard-radiator-your-build-light-2/ [↑](#footnote-ref-32)
33. Viz http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html [↑](#footnote-ref-33)
34. Viz http://pear.php.net/manual/en/standards.php [↑](#footnote-ref-34)
35. Viz http://framework.zend.com/manual/en/coding-standard.html [↑](#footnote-ref-35)
36. Viz <http://cruisecontrol.sourceforge.net/> [↑](#footnote-ref-36)
37. Viz <http://phpundercontrol.org/> [↑](#footnote-ref-37)
38. Viz <http://code.google.com/p/xinc/> [↑](#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://www.urbancode.com/html/products/build/default.html> [↑](#footnote-ref-41)
42. Viz <http://www.jetbrains.com/teamcity/> [↑](#footnote-ref-42)
43. Viz <http://www.jetbrains.com/> [↑](#footnote-ref-43)
44. Viz <http://msdn.microsoft.com/en-us/vstudio/ff637362> [↑](#footnote-ref-44)
45. Viz <http://sismo.sensiolabs.org/> [↑](#footnote-ref-45)
46. Viz <http://travis-ci.org/> [↑](#footnote-ref-46)
47. Viz <http://www.sonarsource.org/> [↑](#footnote-ref-47)
48. Viz <http://docs.codehaus.org/display/SONAR/Hudson+and+Jenkins+Plugin> [↑](#footnote-ref-48)
49. Viz <http://news.cnet.com/8301-30685_3-20000019-264.html> [↑](#footnote-ref-49)
50. Viz <http://jenkins-ci.org/content/whos-driving-thing> [↑](#footnote-ref-50)
51. Viz <http://jenkins-ci.org/content/hudsons-future> [↑](#footnote-ref-51)
52. Viz <http://jenkins-ci.org/content/jenkins>, <http://jenkins-ci.org/> [↑](#footnote-ref-52)
53. Viz <http://kitscm.vse.cz/> [↑](#footnote-ref-53)
54. Viz https://wiki.jenkins-ci.org/display/JENKINS/LTS+Release+Line [↑](#footnote-ref-54)
55. package repository [↑](#footnote-ref-55)
56. Viz https://wiki.jenkins-ci.org/display/JENKINS/Locale+Plugin [↑](#footnote-ref-56)
57. http://localhost:8080/configure [↑](#footnote-ref-57)
58. Viz https://wiki.jenkins-ci.org/display/JENKINS/Green+Balls [↑](#footnote-ref-58)
59. Viz http://pear.php.net/ [↑](#footnote-ref-59)
60. Viz http://www.zend.com/en/products/studio/ [↑](#footnote-ref-60)
61. Viz http://eclipse.org/pdt/ [↑](#footnote-ref-61)
62. Viz http://netbeans.org/features/php/ [↑](#footnote-ref-62)
63. Viz [BERGMANN, 2008] [↑](#footnote-ref-63)
64. Viz http://blueparabola.com/blog/subversion-commit-hooks-php [↑](#footnote-ref-64)
65. Viz http://phpadvent.org/2008/dont-commit-that-error-by-travis-swicegood [↑](#footnote-ref-65)
66. Viz http://framework.zend.com/ [↑](#footnote-ref-66)
67. Vit http://pear.php.net/package/PHP\_CodeSniffer/ [↑](#footnote-ref-67)
68. Viz http://framework.zend.com/manual/en/coding-standard.html [↑](#footnote-ref-68)
69. Viz https://github.com/sebastianbergmann/phpcpd [↑](#footnote-ref-69)
70. Viz https://github.com/sebastianbergmann/phploc [↑](#footnote-ref-70)
71. Viz http://phpmd.org/ [↑](#footnote-ref-71)
72. Viz http://pmd.sourceforge.net/ [↑](#footnote-ref-72)
73. Viz http://phpmd.org/documentation/creating-a-ruleset.html [↑](#footnote-ref-73)
74. Viz https://github.com/sebastianbergmann/phpunit/ [↑](#footnote-ref-74)
75. Viz http://seleniumhq.org/ [↑](#footnote-ref-75)
76. Viz <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html> [↑](#footnote-ref-76)
77. Viz <http://www.phpdoc.org/> nebo <http://pear.php.net/package/PhpDocumentor> [↑](#footnote-ref-77)
78. Viz <http://pear.php.net/bugs/bug.php?id=12128> [↑](#footnote-ref-78)
79. Viz <http://www.beranek.de/node/73> [↑](#footnote-ref-79)
80. Viz <http://www.docblox-project.org/> [↑](#footnote-ref-80)
81. Viz <http://zend-framework-community.634137.n4.nabble.com/ZF-API-documentation-td3451968.html#a3452171> [↑](#footnote-ref-81)
82. Viz <http://docs.docblox-project.org/for-users/validating-documentation-in-your-code.html#as-checkstyle-log-file> [↑](#footnote-ref-82)
83. Viz <http://apigen.org/> [↑](#footnote-ref-83)
84. viz <https://github.com/apigen/apigen/downloads> [↑](#footnote-ref-84)
85. Viz http://ant.apache.org/ [↑](#footnote-ref-85)
86. Viz http://www.phing.info/ [↑](#footnote-ref-86)
87. Viz http://rake.rubyforge.org/ [↑](#footnote-ref-87)
88. Viz http://maven.apache.org/ [↑](#footnote-ref-88)
89. Viz http://nant.sourceforge.net [↑](#footnote-ref-89)
90. Viz http://akrabat.com/zend-framework-tutorial/ [↑](#footnote-ref-90)
91. Viz http://framework.zend.com/ [↑](#footnote-ref-91)
92. Viz https://github.com/mhujer/zf-tutorial [↑](#footnote-ref-92)
93. Viz http://www.phing.info/docs/guide/stable/chapters/appendixes/AppendixC-OptionalTasks.html#PhpLintTask [↑](#footnote-ref-93)
94. Viz https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin [↑](#footnote-ref-94)
95. Viz https://wiki.jenkins-ci.org/display/JENKINS/Phing+Plugin [↑](#footnote-ref-95)
96. Viz https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin [↑](#footnote-ref-96)
97. Viz https://wiki.jenkins-ci.org/display/JENKINS/HTML+Publisher+Plugin [↑](#footnote-ref-97)
98. Viz https://wiki.jenkins-ci.org/display/JENKINS/Plot+Plugin [↑](#footnote-ref-98)
99. Viz https://wiki.jenkins-ci.org/display/JENKINS/JDepend+Plugin [↑](#footnote-ref-99)
100. Viz https://wiki.jenkins-ci.org/display/JENKINS/Violations [↑](#footnote-ref-100)
101. Viz https://github.com/Mayflower/PHP\_CodeBrowser [↑](#footnote-ref-101)