VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

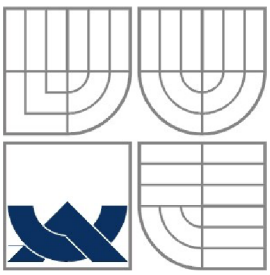
STANDARDY A KÓDOVÁNÍ ZDROJOVÉHO KÓDU PHP

DIPLOMOVÝ PROJEKT
MASTER'S THESIS

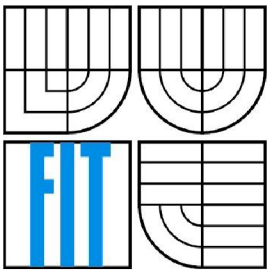
AUTOR PRÁCE
AUTHOR

Bc. Oldřich Pospíšilík

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

STANDARDY A KÓDOVÁNÍ ZDROJOVÉHO KÓDU PHP

STANDARDS AND CODING OF PHP SOURCE CODE

DIPLOMOVÝ PROJEKT
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Oldřich Pospíšilík

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Jaromír Marušinec, Ph.D., MBA

BRNO 2007

Abstrakt

Tento diplomový projekt pojednává o metodice psaní zdrojových kódu a jejich vlivu na efektivitu programování. Dále potom možností odhalování metodických chyb ve zdrojovém kódu PHP. Konkrétně je rozebrána možnost integrace nástroje pro statickou analýzu v rámci vybraného vývojového týmu. Jako vývojový tým byl konzultantem Ing. Michalem Juroszem zvolen programátorský tým, který má na starosti vývoj a rozšiřování internetového informačního systému VUT v Brně. V této prácí jsou uvedeny nejzajímavější současné nástroje pro statickou analýzu jazyka PHP. Po zhodnocení a následném výběru nástroje a dalšího postupu je vypracována analýza a neformální specifikace nástroje. Následuje podrobný návrh, popis implementace a integrace.. V závěru najdeme zhodnocení celé této práce, přínos pro programátorský tým a pokračování vývoje nástroje.

Klíčová slova

PHP, metodiky, chybové vzory, zdrojový kód, statická analýza, transformace

Abstract

This master's thesis deals with the methodology of writing the source code and their impact on the effectiveness of programming. Furthermore, the possibility of error detection patterns in the source code of PHP. Specifically, it addressed the possibility of integration tools for static analysis of the working group. The working group was elected by supervisor Ing. Michael Jurosz, which is in charge of the development and expansion of the Internet Information System Technical University of Brno. The works are given the best tools for static analysis of the PHP language. After evaluation and subsequent selection of tools and the procedure is further analysis and informal specifications tools. The following is a detailed proposal, a description of the implementation and integration .. In conclusion, we find an assessment of the whole of this work, added value for working team and the continuation of development tool.

Keywords

PHP, metodiky, error patterns, source code, static analysis, transformation

Citace

Pospíšilík Oldřich: Standardy a kódování zdrojového kódu PHP. Brno, 2008, semestrální projekt, FIT VUT v Brně.

Standardy a kódování zdrojového kódu PHP

Prohlášení

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením Ing. Jaromíra Marušince, Ph.D., MBA. Další informace mi poskytl Ing. Michal Jurosz.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jméno Příjmení
Datum

Poděkování

Tímto bych chtěl mnohokrát poděkovat Ing. Michalovi Juroszovi za poskytnutí literatury věnující se metodikám psaní zdrojových kódu, za mnoho zajímavých zdrojů, které pomohly rozšířit obsah této práce a vůbec za celkovou spolupráci na této práci

Obsah

1 Úvod

Udržování jednotného stylu programování zdrojového kódu je problémem nejednoho programátorského týmu. Tato práce se snaží teoreticky tento problém rozebrat a zamýšlí se nad zakládáním metodik pro přehledné psaní zdrojových kódů. Tento problém spolu s chybovostí při kódování je důkladně rozebrán v prvních částech práce. Součásti těchto částí jsou i úvahy na téma řešení možných problémů.

V další části je stručně popsána pracovní skupina, pro kterou bude vyvíjena v rámci této práce sada nástrojů pro statickou analýzu zdrojových kódů. Konkrétně se jedná o část oddělení vývoje. CVIS (Centrum Výpočetních a Informačních Služeb), která má na starosti vývoj webových aplikací. Problémy tohoto týmu a potřeby mi byly sdělovány Ing. Michalem Juroszem, jenž pracuje v oddělení vývoje a je plně seznámen s aktuálními potřebami tohoto velkého projektu. Některé z nich by tato práce měla uspokojit. Součásti zmíněných kapitol je stručný popis této aplikace, na kterou budou nástroje a získané teoretické poznatky aplikovány.

V následující kapitole se práce podrobně věnuje problematice výše zmíněné statické analýzy. Nejdříve si dovoluje statickou analýzu rozdělit a dále navázat na jeden z typů, který je podrobněji rozebrán. Na konci této kapitoly jsou zhodnoceny současně dostupné nástroje pro statickou analýzu. K aplikacím je práce velmi kritická a zkoumá jejich možnosti uplatnění v rámci pracovní skupiny a zejména si všímá možnosti jejich rozšíření. Mezi analyzované nástroje patří PHP Security Scanner, Pixy, PHP Sat, PHP Front a PHC. Zmíněna je i možnost vývoje vlastního nástroje.

Dále je detailně rozebrán nástroj PHC. Nejdříve je analyzováno, jakým způsobem tento nástroj pracuje. Samostatná kapitola je věnována jakým způsobem PHC pracuje s gramatikou jazyka PHP. Následují podkapitoly popisují aplikační rozhraní PHC, které umožňuje pomocí C++ kódu pohodlně procházet a případně transformovat syntaktický strom zdrojového kódu PHP.

Zbytek práce se věnuje vlastnímu nástroji, který koresponduje s touto prací. Kapitola je též věnována podrobné analýze požadavků, které byly za dobu práce na teoretické částí této práce shromážděny. Následuje detailní popis návrhu a následné implementace. V předposlední kapitole je nástroj podroben integračnímu testování na reálné aplikaci přiřazeného vývojového týmu.

V závěru jsou lehce nastíněna možná pokračování této práce. Následuje celkové zhodnocení této práce . Zmíněny jsou i názory programátoru na význam této práce.

2 Problematika kódování

2.1 Psaní zdrojového kódu

2.1.1 Psaní čitelného kódu

Mnoho programátorů se domnívá, že míra dovednosti programování je nejvíce ovlivněna rychlostí, za jakou je programátor schopný daný problém algoritmicky vyřešit. Dokonce se programátoři předhání, kdo dokáže problém implementovat, na co nejméně řádku kódu. Používají složité výrazy se spoustou vedlejších efektu, které ve výsledku někdy až neuvěřitelně zkrátí programový kód. Typickým příkladem je jazyk C a jazyky, které mají odvozenou syntaxi. Programátoři mají pocit, že jejich skutečně textově krátký zdrojový kód bude velice rychlý, a že délka zdrojového kódu je přímo úměrná procesorovému času. Opomineme-li některé techniky pro optimalizaci rychlosti kódu, což není tento případ, není tento pocit ani trochu pravdivý. Dnes již jsou optimalizátory v překladačích na takové úrovní, že jim nemá smysl „radit“ a takovýmto způsobem ulehčovat práci. Jak takový kód může vypadat v jazyce C :

```
while(*dist++ = *source++);
```

Elegantní? Ano, ale pokud se na tento kód podívá jazyka C neznalý programátor, kolega jenž zdrojový kód chce použít, musí vynaložit poměrně velké intelektuální úsilí pro pochopení tohoto řádku.

Dostáváme se k tomu, že kód by měl být srozumitelný nejen překladači, což vzhledem k pokročilým vlastnostem kompilátorů a jejích chybovým výstupům není žádný problém, ale hlavně lidem. Zdrojový kód, kterému nikdo nerozumí nebo k jeho pochopení bude potřebovat stejně nebo víc času, jako k napsání jiného algoritmu řešící stejný problém, nemá smysl psát. Praktický žádný kód není na jedno použití a nikdy člověk neví, komu se zdroj dostane do rukou a jak jeho rozvoj bude pokračovat.

Často bývá nečitelnost kódu využívána, jako forma ochrany intelektuálního vlastnictví. Programátor využívá nečitelnosti svého výtvoru k ochraně proti zneužití. Takovéto uvažování není naprosto scestné, jen se k němu musí programátor postavit jinak. K ochraně duševního vlastnictví jsou určeny zákony daného státu a pokud již někdo chce využít nečitelnosti kódu, jsou k dispozici nástroje, které umožňují maskovat kód. Ve výsledku by tedy mělo platit, maskovat přehledný kód a maskovat jen jeho verzi určenou k nasazení v reálné praxi. Využívá se mazání tabulátorů, mezer a komentářů. Pro příklad v jazyku PHP se často využívá zakódování zdrojového kódu a jeho vyhodnocení pomocí vestavěných funkcí. Přesto lze pomocí nástrojů reverzního inženýrství a trochou kreativity převést kód zpět do čitelné podoby.

2.1.2 Komentování zdrojového kódu

Komentáře tvoří základ čitelného, správně napsaného kódu a jsou nejlepší dokumentací k projektu. Rozhodně platí, že nedostatek komentářů je větší chybou než jejich přebytek. Měli by být stručné, jasné a neměli by duplikovat ostatní části dokumentace, jako například referenční příručku. Chybou také je, když komentáře duplikuje činnost kódu. Takový komentář nám nic nového neposkytne a může jen zastiňovat užitečný a podstatný komentář. Nepřímo se pomocí komentářů odehrává komunikace mezi členy týmu. Dobře okomentovaný zdrojový kód je lehce udržovatelný a komunikace v týmu se omezí jen na podstatné problémy. V neposlední řadě pokud se programátor vrátí po delší době ke svému kódu, komentáře mu umožní se ve své práci rychleji zorientovat.

Komentovat je vhodné bloky kódu, které jsou pak lehce čitelné a přeskočením bloku lze pokračovat ve čtení kódu bez sebemenšího zmatení. Jednotlivé bloky kódu mají připomínat věty, které se pak snadno čtou a výsledný program či část lze snadno pochopit. Komentovat by se měly místa, která jsou obtížně pochopitelná a nejasná. Je třeba být na pozoru, co je srozumitelné pro autora nemusí být srozumitelné pro ostatní, kteří se také potřebují v kódu vyznat. Je zvykem komentovat definice nových datových struktur a definice procedur či funkcí. Pokud se rozhodneme použít nějaký „trik“ pro zrychlený zápis (viz kapitola 1.1.1), je třeba ho taktéž okomentovat. Někdy se komentují i invarianty cyklů. Zvláštním případem cyklení kódu je rekurze, která může být obzvlášť matoucí a zaslouží si dostatečné okomentování.

Některá vývojová prostředí dokážou pomocí vhodně formátovaných komentářů vygenerovat automaticky technickou dokumentaci nebo nabízet programátorům vlastnosti a metody již s popisem, kde komentáře plně nahrazují nápovědu. Často se v těchto komentářích používá syntaxe značkovacího jazyka XML. Týmová práce v takto vedených zdrojových souborech dostává nový rozměr, když každý člen týmu nebo případně člověk, který chce zdrojový kód využít, dostává spolu s kódem i nápovědy, jenž mu vývojové prostředí přehledně ukazuje při jeho práci. Problémem zůstává velké množství prostředí, které každé používá trochu jiný formát komentářů a navzájem si nerozumí. Programátorovi používající jiný nástroj tak formátované komentáře fungovat nebudou, ale přínos zde samozřejmě stále existuje.

Jazyk komentářů nemusí být nutně angličtina. Pokud není celý tým dostatečně jazykově zdatný a pochází ze stejné neanglicky mluvící země, bude angličtina jen na přítěž. U mezinárodních týmů je nutné zvolit společný jazyk. Komentáře by totiž měly být napsány v rámci projektu stejným jazykem. Někdy mohou být neanglické komentáře využívány k jisté formě maskování kódu (viz konec kapitoly 1.1.1). Vzhledem k tomu, že například čeština není po světě zrovna nejrozšířenějším jazykem, značně zredukujeme počet programátorů chápající naše komentáře.

Zvláštní případem jsou komentáře hlaviček zdrojových souborů. Často v nich nalezneme informace o časech založení modulu, jeho poslední úpravě a informace o majiteli, případně kontakt na něj či typ licence. Mnoho těchto atributů jsou při používaní systému pro správu verzí a projektu

prakticky nepotřebné. Přesto by se tým měl dohodnout na formátu a atributech těchto hlaviček. Prakticky jediným požadavkem je, aby hlavičky v jednom projektu měly tento formát jednotný.

2.1.3 Pravidla pro čitelný kód

Komentáře nejsou jediným prostředkem pro čitelný a přehledný kód. Mezi základní techniky čitelného kódování je rozumný formát odsazení a uspořádání zdrojového textu. Odsazení by se mělo volit mezi velikostmi dvou až pěti mezer, či jednoho tabulátoru. Konkrétně je třeba se věnovat pozornosti rozlišení počítače každého člena týmu. Vybrat nejmenší a následně zvolit odsazení takové, aby pokud možno žádná část zdrojového textu nezasahovala za okraje obrazovky.

Další důležitým prostředkem pro zpřehlednění kódu je odsazování smysluplných bloků kódu prázdnými řádky. Zdrojový kód je pak velmi dobře čitelný po jednotlivých blocích. Pokud je blok používán častěji, je vhodné uvažovat o definici nové funkce nebo procedury. Počet řádku procedury by neměl přesáhnout počet v rámci jedné stránky editoru. Dekompozice, ale již patří k základním programátorským dovednostem, a proto toto téma zde nemá cenu více rozebírat.

Samostatným problémem je pojmenování identifikátorů. Je všeobecně dobrým programátorským zvykem psát identifikátory pět až dvanáct znaků dlouhé a měly by pojmenováním říci k jakému účelu byl vytvořeny. Delší pojmenování je vhodné použít jen u identifikátorů s menší frekvencí výskytu a jen pokud je přínosem větší přehlednost. V jazycích „case sensitive“, kde záleží na velkosti písmen se velikosti využívá k rozdělení slov v identifikátorech. Taktéž velikosti písmen bývá využíváno k určení přístupnosti proměnných a to tak, že proměnné začínající malým písmenem jsou privátní a naopak proměnné začínající velkým písmenem jsou definovány jako veřejné.

V praxi je snad nejviditelnější používání předpon u identifikátorů. Když se například správně nekontrolovala v překladačích typovost, často se používala předpona identifikátoru k určení datového typu. V dnešní době tento programátorský rys už upadá a předpony se využívá k určení přístupnosti nebo k určení zda se například jedná o globální proměnnou nebo konstantu.

Volba jazyka pro pojmenování se od volby u komentářů mírně odlišuje, protože nároky na jazykovou úroveň programátorů není tak vysoká a neanglické jazyky často obsahují pro identifikátory zakázané znaky. Angličtina proto bývá většinou tou správnou a nejčastější volbou. Výhodou neanglického pojmenování se může jevit opět jako jistá forma maskování. Podobně, jak bylo zmíněno v kapitole 1.1.2.

2.1.4 Konvence kódování a pravidla

Z předchozích kapitol vyplývá, že kromě toho, aby kód byl pro člověka dobře čitelný je dalším cílem, aby zdrojový kód vypadal, jako by ho psal jeden člověk i když na něm pracuje celý tým.

V rámci týmu je nejlépe před samotným zahájením projektu nebo v jeho začátcích zvolit konvenci pojmenování identifikátoru a určit pravidla pro přehledný a čitelný kód. Není proto divu, že

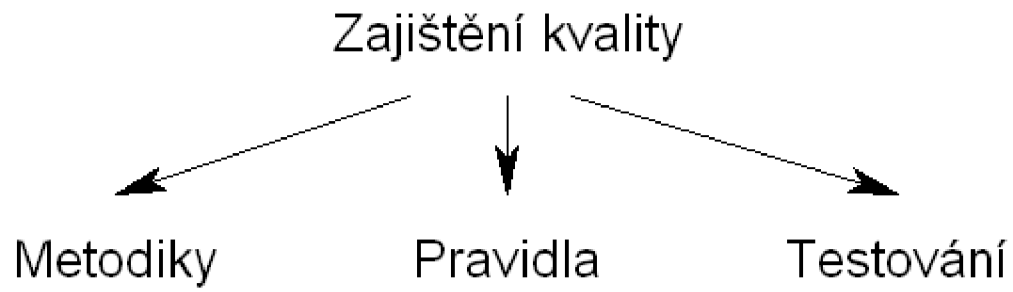
ve velkých projektech se konvence dostaly mimo konkrétní tým a rozšířili se do obecného povědomí programátorů. Maďarská konvence například obsahuje u identifikátorů předponu, která určuje datový typ. Maďarská konvence vznikla ve firmě Microsoft a na jejich stránkách je ještě v této době k nahlédnutí. Známá je i takzvaná „velbloudí“ konvence využívající malých a velkých písmen.

2.2 Chyby zdrojového kódu

2.2.1 Co je to chyba

Pro pozdější definici statické analýzy je třeba hlouběji se zamyslet nad chybami ve zdrojovém kódu a jejich případným rozdělením. Chyby patří k programování a je třeba s nimi bojovat od samého začátku vývojového procesu. Diky komplexní struktuře kódu může jedna chyba nabalovat jinou a oprava, která přijde později může znamenat tři další. Nakonec vývojáři zjistí, že se jim celý projekt pod rukama rozpadá a je nutné odsunout termín jeho vypuštění do reálného provozu. Nejsou žádnou výjimkou projekty, při nichž množství chyb znamená stop a projekt, jak se říká „umře“ a vývoj ustane.

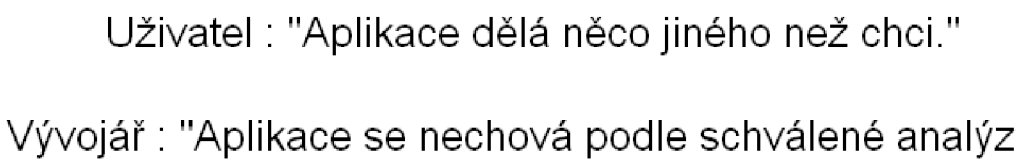
Chyby úzce souvisí se zajištěním kvality (z anglického „quality assurance“), což je proces (viz obrázek číslo 1), který si nese za cíl minimalizovat chyby v hotovém projektu.



Obrázek 1 : Zajištění kvality

Metodika popisuje pravidla, která mají vést ke snížení rizik a tím i následných chyb. Testování má za úkol odhalení chyb, na které metodiky a pravidla nedosáhly.

Na definici pojmu chyba se můžeme podívat ze dvou hlavních stran. Ze strany vývojáře a ze strany uživatele (obrázek číslo 2).



Obrázek 2 : Co je to chyba

Jak popisuje obrázek, pro vývojáře je chybou cokoliv, co se odkloňuje od analýzy schválené, podepsané zákazníkem a vedoucím projektu. Zde ovšem konflikt vývojářů, kteří díky nejednoznačnému nebo neúplnému zadání vyrobili chybu a nejsou si toho vědomi. Naopak pro uživatele je pohled na chybu naprosto odlišný. Podle nich je chybou cokoliv, co není podle jejich představ a vnímají to negativně. V jejich případě může dojít také ke konfliktu, kdy si uživatel nebyl všech rizik a vůbec podoby programu vědom a objeví chybu později. Vývojáři s ní v analýze samozřejmě vůbec nepočítali. Takovým chybám se snaží předejít různé metodiky vývoje softwarových projektu, ale to už je mimo rámec této práce.

2.2.2 Rozdělení chyb

Pro pozdější analýzu je třeba si chyby v kódu rozdělit a klasifikovat. Mezi základní typy chyb patří technologické nedostatky, programátorské chyby, nedostatky v zadání a ostatní chyby. Dále se tímto základním dělením zabývat práce nebude a vezme v úvahu jen chyby programátorské.

Programátorské chyby můžeme rozdělit na syntaktické a sémantické. Syntaktické chyby jsou nejjednodušší na vytvoření, ale taky na odhalení. Syntaktický chyba je vše, co způsobí nesprávný kód, který nejde přeložit (jinak řečeno se jedná o prohřešek proti pravidlům gramatiky jazyka). Parser překladače si s ním neporadí a překlad skončí chybou. Překladač nám v naprosté většině dá hláškou vědět o přesném umístění chyby. Další typ programátorských chyb jsou chyby sémantické. Sémantická neboli významová chyba se neprojeví při překladu a je větší problém ji najít. Tyto chyby způsobují nesprávnou funkci programu a k jejímu odhalování se používají včasně a správně navržené aplikační testy.

Chyby můžeme z druhého pohledu dělit podle závažnosti nebo oboru, kterého se týkají. Tato rozdělení už jsou čistě v rámci jednotlivých nástrojů, které jakýmkoliv způsobem pracují s programátorskými chybami. Pro příklad můžeme klasifikovat chyby na vláknové (vznikají nesprávným použitím nástrojů pro správu vláken, jako prostředku pro paralelní programování), na chyby v zabezpečení a podobně.

2.2.3 Hledání chyb během vývoje

Jak již bylo zmíněno hledání a odstraňování chyb má největší význam během vývoje. Abychom se vyhnuli kumulování chyb. Existují celkem tří programátorské techniky pro hledání a objasňování chyb. Většinou bývají podporovány vývojovým prostředím.

Ověřovací podmínky (z anglického „asserts“) jsou podmínky, o kterých mlčky předpokládáme, že budou za všech okolnosti správné. Pokud tyto podmínky nejsou platné, většinou dojde k pádu aplikace. Je třeba takové podmínky ověřovat v kódu co nejvíce a vyvolávat výjimky či dialogová okna s informací, případně přidat záznam do souboru se záznamem událostí. Každá taková podmínka zpomaluje výsledný program a ve výsledné podobě není nutná, protože prakticky odhaluje chyby v programu, které ve výsledné verzi nebudou. Dalším problémem je, že nemusíme vědět, která z reakcí na podmínku bude ta správná. Nejlepší je, když je možnost nastavit toto chování bez nového překladu. Například pomocí parametru při spuštění. Často je nutné takové chování dosáhnout standardními programovacími prostředky. Některá vývojová prostředí dokážou programátorům ušetřit práci a obsahují již implementaci takových prostředků. Pro příklad příkaz „assert“ v prostředí .NET od firmy Microsoft.

Další technikou je záznam událostí (z anglického „logging“). Při tzv. „logování“ se všechny důležité, neobvyklé nebo chybové události ukládají do externího souboru, databáze nebo jiného úložiště. Na rozdíl od ověřovacích podmínek, běží záznam událostí i v „ostré“ verzi, a je proto velmi důležitým nástrojem pro zjišťování chybových stavů systému během zkušebního provozu. Do záznamu se ukládá informace o chybě, datum, čas a někdy i stav systému, v kterém se v dané době nacházel. Při této technice je důležité rozlišovat o jak důležitý záznam se jedná. Často je možné v jednotlivých fázích vývoje vypínat některé úrovně výstupů.

Trasování (z anglického „tracing“) je výhodné pokud řešíme výkonnostní problém a potřebujeme podrobný výpis popisující chování určité části aplikace. Trasování by se dalo popsat, jako podmíněný záznam událostí.

Z jiného pohledu hledání chyb během vývoje můžeme k technikám zařadit i analýzu kódu. Tato technika je ale více náročná na konfiguraci a zavedení do projektu, proto je využívána jen u větších projektů. Analýza nenachází chyby sama o sobě, ale pomocí ní je možné hledat chybové vzory již známých typů chyb. Výhodou analýzy je, že vzhledem k její velmi značné modifikovatelnosti je často možné ji rozšířit o automatické nahrazovaní nalezených metodických chyb. Chyby tak odstraňuje prakticky automaticky. Více o analýze, u které použití k hledání chyb během vývoje zdaleka nekončí a patří spíše mezi minoritní možnosti uplatnění této techniky, v kapitole 3.

2.3 Metodik

2.3.1 Metodik jako člen vývojového týmu

Popis pravidel a předpisů pro čitelný zdrojový kód a pro případnou detekci chyb v týmu dohlíží a současně navrhuje vedoucí projektu. V případě menších projektů pod záštitou nevelkých týmů se tato činnost neprávem úplně zanedbává. V rámci větších a ambicióznějších projektů je velmi často zvolen člověk, který má metodiky na starosti. Předchozí věty budou brány, jako definice osoby metodika v rámci této práce, protože se v oblasti různých profesí nebo projektů může tato definice mírně lišit. Metodik nejdříve sleduje svou pracovní skupinu v rámci dřívějších projektů a následně společně s nimi sepíše metodiky, které bude tým dodržovat. Metodik často velmi přehání a je dobré dávat pozor, aby zmíněná pravidla nepřinesla více škody než užitku. Práce metodika není prakticky nutná po celý životní cyklus softwarového projektu a po zavedení nástrojů pro udržování, kontrolu pravidel již tato funkce není nutná nebo má velmi málo práce. Metodik proto mění svou funkci v týmu a věnuje se jiným činnostem.

Obsah této práce je v podstatě teoretická příprava metodika a následná implementace vyhodnocených pravidel do reálné praxe na určitý projekt. ¨

2.3.2 Práce metodika

Při založení nového projektu je metodik prověřen vytvoření pravidel. Formou rozhovoru nebo dotazníků sepíše požadavky na formát zdrojového kódu, připomínky k již hotovým projektům jednotlivých programátorů a základní informace o jejich práci. Následně posbírá ukázky zdrojových kódu programátoru a podrobí je vizuální analýze. V poslední řadě podrobí analýze nástroje, které používají programátoři ke své práci. Zde je tím myšlena jak již hardwarová část, rozlišení obrazovek a možnost síťové komunikace tak také softwarová část v podobě vývojových prostředí a editorů.

V tuto chvíli by již měl mít posbírán dostatek informací k tomu, aby začal sepisovat metodiky psaní zdrojového kódu. Po sepsání prodiskutuje metodiky opět s programátory. Pokud jsou metodiky hodnoceny záporně, cyklus se opakuje a v opačném případě jsou metodiky uvedeny do praxe.

V období vývoje a používaní metodik musí být znám způsob, jakým se budou metodiky kontrolovat. Jsou možné externí kontroly pověřenou osobou, vzájemné kontroly kolegů a nejobvyklejší jsou kontroly pomocí nástrojů, které lze lehce automatizovat..

2.3.3 Nástroje metodika

Jako hlavní nástroje pro dodržování metodik jsou aplikace pro statickou analýzu zdrojového kódu (viz následující kapitola). Kromě zmíněných vlastností dovolují daleko širší manipulaci se zdrojovým kódem. Tyto techniky transformace kódu budou v této práci dále rozvedeny.

2.4 Úvod do analýzy zdrojového kódu

2.4.1 Co je analýza kódu

Analýza kódu je široký pojem, při němž pracovník většinou automatizovaně, pomocí nástroje analyzuje funkce a složení kódu. V základě se analýza dělí na statickou a dynamickou. O dynamické zde bude zmíněna jen jedna krátká kapitola, přičemž statické analýze bude věnován prakticky celý zbytek práce.

2.4.2 Dynamická analýza kódu

Dynamická analýza dovoluje na rozdíl od statické kód spustit a následně jej analyzovat. To je velmi výhodné, pokud chceme například sledovat a rozebírat práci s pamětí či vlákny a vůbec problémem spojenými s paralelním programováním. Jen pro zajímavost zde patří nástroje ConTest, CCuret, Pyrity a snad nejpoužívanější Valgrind.

2.4.3 Statická analýza kódu

Statická analýza naproti tomu rozebírá kód v jeho čisté podobě a kód není spuštěn. V některých jazycích je analyzován mezikód, ale jedná se spíše o výjimky. Problémy čitelného zdrojovém kódu a chybám v něm byli věnovány předchozí kapitoly. To vše umí řešit nástroje pro statickou analýzu kódu. K nejznámějším nástrojům pro tento typ analýzy patří PMD a FindBug pro jazyk Java. Pro jazyk C existuje sada komerčních nástrojů CodeSonar.nebo Astre.

2.5 Pracovní skupina

2.5.1 Popis pracovní skupiny a projektu

Jako pracovní skupina, ve které budou zaváděny nové metodiky a prováděna pravidelná analýza kódu, byl vybrán vývojářský tým oddělení vývoje CVIS, který má na starosti vývoj webových aplikací. Tým má nestálý počet členů a na vývoji se podílí řada externistů. Daleko více, než v jiném týmu zde tedy vyniká neucelenost kódu a individuální programátorský styl. Těmto neduhům pak nahrává, že tým se prakticky věnuje jednomu velkému projektu po celou dobu (zanedbáme-li jeho rozdělení na moduly). Je jasné, že možné chyby spojené s nejednotností mohou vést k chybám, které se svou vážností nahromadí a nikomu známe závislosti způsobí vznik chyb dalších.

Projekt je napsán v jazyku PHP, HTML a JavaScriptu. Současně se používá jazyk PHP ve verzi 5. Metodiky a nástroje budou vyvíjeny jen pro jazyk PHP, kterým se píší veškeré moduly. Díky

tomuto faktu bude jazyku PHP věnováno v této práci dostatek místa a na volbě nástroje bude tato skutečnost hrát zásadní roli.

V rámci této práce bude nejdříve skupina sledována, analyzována a budou navrhnuty metodiky, které se v závěru bude metodik snažit uvést v praxi. Během práce na zjišťování nejefektivnějších metodik bude vybrán vyhovující nástroj pro statickou analýzu, implementován vlastní nebo v třetím případě vznikne nový nástroj kombinací již existujícího.

Z dosavadního popisu je jasné, že projekt se nachází již ve velmi pokročilé fázi životního cyklu, kdy již je integrován do provozu a pracuje se na nových částech, vylepšují se již hotové a přepisují zastaralé. Proto je důležité při volbě nástroje myslet na to, že je třeba již hotový kód převést a řádně, pomocí přehledného výstupu, informovat o jeho stavu. Aplikace je velmi rozsáhlá, proto je prakticky nutností automatizovat, co největší počet oprav a zamezit tak vzniku nových chyb a zdržení vývojového týmu. Těmto a dalším úskalím budou věnovány další kapitoly při analýze požadavků.

2.5.2 Současná metodika skupiny

Současné metodiky zdrojového kódu této skupiny jsou založeny na obecných pravidlech, které se vyvinuly z obecných potřeb programátorů a jejich předvídavosti. Nejsou nikde psány a neexistuje dokument, který by je do detailu popisoval. Metodiky na obecné domluvě fungují do té doby, než je někdo poruší. Metodiky současné budou zachovány, formálně definovány a následně přidány nové. Nástroj se v neposlední řadě postará o včasné zachycení nedodržování metodik a informování viníka.

2.5.3 Možné přínosy ve skupině

Již v této době kód začíná být nepřehledný. Pro příklad v této době v celé aplikaci existují celkem tři připojení k databázi a přitom stačí když bude funkční je jen jedna Kód by tedy měl být vyčištěn od zbytečných konstrukcí a je jisté, že tyto úpravy budou mít kladný vliv na rychlost celé aplikace.

Díky zásahům do metodiky programování kódu se počítá, že se zpřehlední dosavadní a budoucí práce. Pokud se metodiky podaří správně aplikovat dojde i ke zrychlení práce díky efektivnější komunikaci. To vše díky jednotnému vzhledu kódu.

Neméně důležitým přínosem bude možnost detekce nejčastějších chyb a jejich snadná pokud možno automatická oprava. Ve zkratce jde tedy o celkové zkvalitnění kódu.

3 Statická analýza

3.1 Rozdělení statické analýzy

3.1.1 On-the-fly analýza kódu

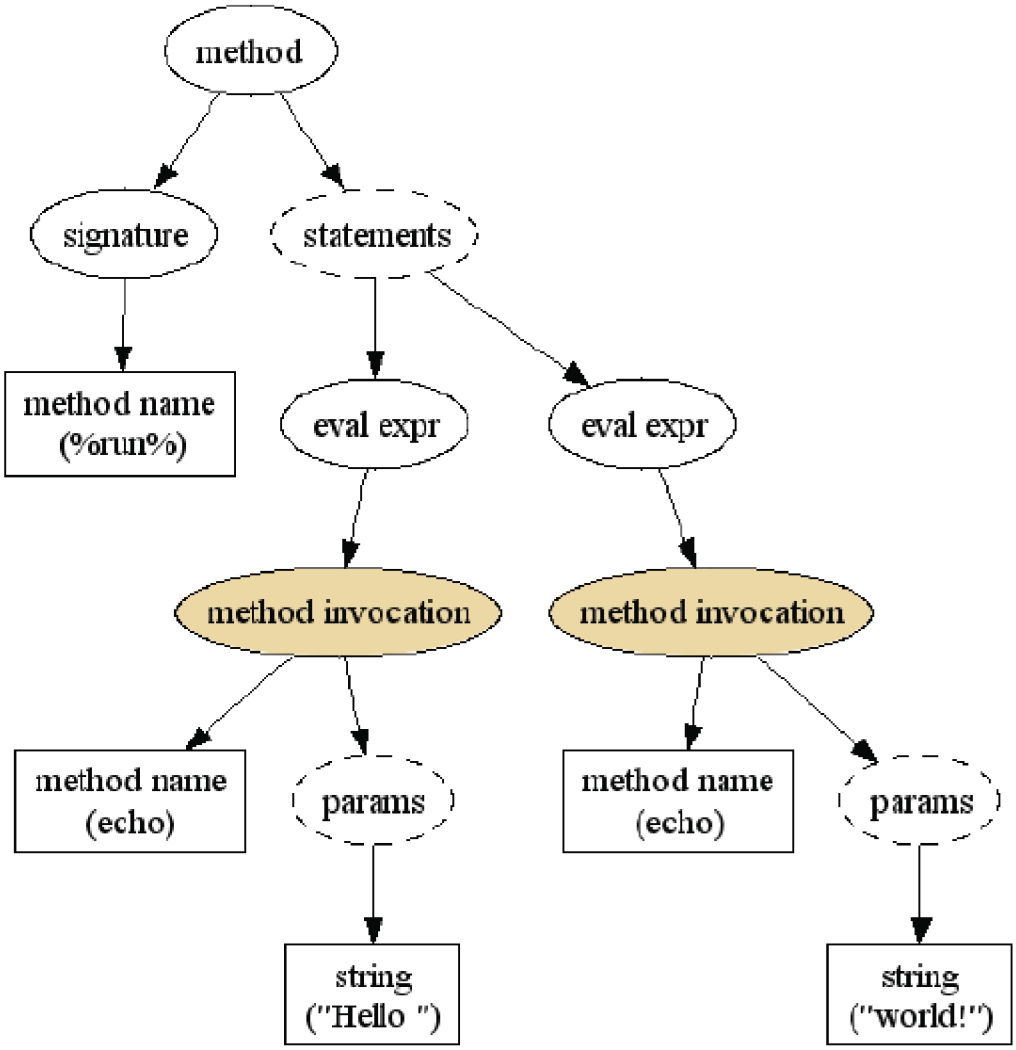
Tento typ analýzy kódu je spouštěn neustále bez žádosti programátora. Dnes ji používá téměř každý editor zdrojového kódu. Díky ní nás prostředí může už během psaní kódu upozorňovat zbarvením kódu a podobnými mechanizmy na nevhodný nebo chybný kód a nabízet jeho automatické opravy (tzv. „click and fix“). Nastavení vyhledávání chybových vzorů není většinou v tomto typu analýzy nastavitelné a je přednastaveno jen na základní sadu chyb. Dále se tato práce tomuto typu analýzy nebude věnovat.

3.1.2 On-demand analýza kódu

Jak již z názvu vyplývá, tento druh analýzy se spouští na žádost. Jde o daleko více pokročilejší techniku než v případě analýzy za běhu. Často obsahuje velice mnoho nastavení a možnost přizpůsobení k aktuálnímu účelu analýzy. Na rozdíl od předchozího typu analýzy, který je vždy součástí nějakého editoru, analýza na žádost bývá často samostatným programem. Často jsou tyto nástroje distribuovány jako „pluginy“ (integrovatelné do vývojových prostředí jenž poskytují grafický výstup a pohodlné uživatelské rozhraní). Z předchozích vět vyplývá, že bývá daleko výpočetně náročnější. Pokud bude dále v práci zmíněná analýza kódu, bude se jednat už vždy jen o tento typ.

3.2 Základy do statické analýzy

Analýza zdrojového kódu je poměrně odlišná od analýzy klasického textu, ale má společné znaky. V každém případě je nutné znát definici gramatiky jazyka, ve kterém je kód napsán. Pokud nedisponujeme takovouto znalostí, nemůžeme zdaleka analyzovat zdroj v takové míře v jaké potřebujeme. Dále tedy budeme uvažovat, že gramatiku jazyka známe. Abychom dostatečně znázornily strukturu kódu, je třeba převést zdroj do datové struktury, se kterou již pak bude lehké dělat analytické a substituční operace. Touto strukturou je syntaktický strom. Příklad takového stromu je na obrázku č.3. S takovouto strukturou již lze pracovat prakticky bez omezení. Velmi lehce vyhledávat a provádět transformace bez možného porušení všech závislostí a tedy struktury kódy.



Obrázek 3 : Syntaktický strom v nástroji PHC

Jako každá analýza i analýza zdrojového kódu něco hledá a zkoumá. Nejdříve je nutné zvolit správný formát, ve kterém bude hledání nějak formálně definováno. Jako ideální se určitě vybaví regulární výrazy. Ponecháme-li stranou jejich výrazovou mocnost, tak se větší nevýhodou bezesporu je to, že pro navigaci po syntaktickém stromu nejsou příliš vhodné. V případě že máme dobře zvolený XML dokument, může být definice hledaného vzoru v XML formátu, ale tato možnost není příliš vhodná a prakticky se nepoužívá. Další poměrně unikátní možností je formát XPath, který díky své navigační stromové struktuře slouží účelu vcelku dobře a skutečně několik málo nástrojů jej používá. Nejpoužívanější možností je definice v objektovém návrhu, kde pomocí dědičnosti a již implementovaných metod je možné pohodlně procházet syntaktický strom. Tuto možnost najdeme prakticky ve všech nástrojích pro statickou analýzu. Později v navazující práci bude formátu prohledávání stromu věnována samostatná kapitola.

Obecnou nevýhodou takového hledání je, že dochází velice často k takzvaným „false alarms“, což jsou vyhledaná místa ve zdrojovém kódu, se kterými vývojář již neplánuje nic dělat a prakticky je ani nechtěl vyhledat. Jedná se o výjimky vyhledávácího pravidla. Aby vyhledávací pravidlo nemuselo být pro každou výjimku měněno, nástroje pro statickou analýzu většinou poskytují techniku, jak již v případě opětovného hledání vzorů výjimky nezobrazit. Nejčastěji se jedná o speciální komentář, který je vkládán před konstrukci. Výhodou následně je, že takový komentář je možné napsat ještě před samotným vyhledáním a nečekat na nástroj, který by vám tento komentář nabídnul.

V případě nálezu v syntaktickém stromu už probíhá akce definovaná ve vyhledávacím pravidle. Zde má obrovskou výhodu implementace pravidla v objektovém návrhu, kdy můžeme akci psát pomocí programovacího jazyka rovnou uvnitř pravidla. Jako akce může být vypsaná hláška, zápis do souboru, přidání statistického záznamu či pokročilejší akce, jako jsou různé varianty transformace syntaktického stromu (vyjmutí, připojení větve a podobně).

Na závěr této kapitoly příklad (na obrázku č.4), kde chceme dosáhnout změny funkce připojení k databázi (nahradit vstup za výstup). Pokud se nad problémem hlouběji nezamyslíme, mohl by někdo říci : „Proč nepoužít klasickou náhradu textu, podporovanou snad všemi textovými editory?“. Odpověď by byla velice rychlá. Náhrada by byla naprosto nevhodná, protože jako argumenty figurují proměnné. Regulární výraz by problém řešit mohl, ale kdybychom přidali podmínku, že ji chceme nahradit jen tam, kde bude jako připojení jako první příkaz ve funkci, nebyl by regulární výraz vůbec vhodnou alternativou, respektive by musel být rozšířeno další funkční kód.

´vstup : mysql_connect (server, username, password, new_link, int client_flags)

výstup : dbx_connect (module, host, database, username, password, persistent)

Obrázek 4 : Příklad hledaného vzoru a jeho nahrazení

3.3 Formáty vyhledávacích pravidel

3.3.1 Objektový návrh

Nejčastější formou definování nových vyhledávacích vzorů je prostřednictvím objektového návrhu. Využívá se dědičnosti, již implementovaným tříd a metod k procházení syntaktického stromu. Zpravidla se nové pravidlo implementuje definicí vhodně pojmenované nové třídy, která dědí ze třídy, která dokáže procházet vhodně strom pomocí svých metod. Tato bázová třída často obsahuje slova jako „rule“ a „tree“ (např. AbstractRule nebo TreeVisitor). Pomocí metod je možné dostat se ke konkrétním prvkům stromu tedy konstrukcím kódu. Pomocí argumentů těchto metod je následně možná transformace stromu nebo nějaké konkrétní porovnání s listem stromu. Na obrázku č.5 je pro

příklad definice nového pravidla nástroje PHC. Jde o nástroj napsaný v jazyku C a je určen pro jazyk PHP. Tomuto nástroji bude věnována později samostatná kapitola. Nová třída „Pocitej“ umí díky dědičnosti procházet strom a počítá počet metod s názvem „foo“.

```
class Pocitej :   public Tree_visitor
{
    private int count = 0;
    void pre_method_name(Token_method_name* in)
    {
       if(*in->get_value == "foo")
            this->count++;
    }
};
```

Obrázek 5 : Ukázka definování nového pravidla. Nástroj PHC.

Každý nástroj má pochopitelně vlastní bázovou sadu tříd, metod a právě z toho vyplývá určitá nevýhoda. Vývojáři, kteří chtějí tuto aplikaci uvést do praxe by měli znát jazyk, ve kterém je implementován nástroj a ten je často jiný než jazyk analyzovaný.

3.3.2 XPath

Použití jazyka XPath pro definování vyhledávacích vzorů v nástrojích pro statickou analýzu je poměrně vzácné. Nejznámějším nástrojem, který ho používá je PMD pro jazyk Java. XPath (XML Path Language) je jazyk, pomocí kterého lze adresovat části XML dokumentu. Výsledkem nějakého Xpath výrazu je nějaká množina uzlů a toho se využívá při vyhledávání syntaktickým stromem. Syntaktický strom lze poměrně snadno převést na XML dokument, který ekvivalentně popisuje zdrojový kód. Jako příklad (obrázek č.6) je uveden příkaz v nástroji PMD, který prohledá strom a vrací metody, které mají více než jednu proměnou datového typu „foo“.

```
TypeDeclaration[count(//VariableDeclarator[../Type/Name[@Image='foo']])>1]
```

Obrázek 6 : Ukázka definování nového pravidla pomocí jazyka XPath. Nástroj PMD.

Z obrázku je vidět, že XPath je velice elegantní a může být daleko rychlejším prostředkem pro vyjádření vzoru jenž hledáme. Přesto pro náročnější vzory není příliš vhodný a bývá doplněn možností vytvořit vyhledávací pravidlo pomocí objektového návrhu v některém z programovacích jazyků.

3.3.3 XML

Značkovací jazyk XML je v nástrojích pro statickou analýzu použit spíše doplňkově a to pro definici souboru pravidel. Každý nástroj má pochopitelně svoje schéma takových XML dokumentů, ale jejich formát je vždy velmi intuitivní a neobsahuje mnoho značek.

3.3.4 Klasické programátorské techniky

Kromě výše zmíněných formátů, které mohou vyhledávání značně ulehčit, je možné vyhledávat pomocí standardních programátorských technik. Pro příklad jde o vyhledávání za pomocí regulárních výrazů nebo lze pomocí lexikální analýzy rozdělit kód na lexémy (tokeny) a dále se již programově navigovat po rozparsovaném zdrojovém kódu.

Některé jazyky podporují pomocí vestavěných funkcí podporu pro lexikální analýzu. Jazyk PHP není výjimkou a obsahuje funkci „get_token_all“, která vrací pole lexémů spolu s čísly řádek jejich výskytů. V dalších kapitolách bude takový přístup důkladně zvážen.

3.4 Analýza ve skriptovací, jazyce PHP

3.4.1 Stručný popis jazyka PHP

Protože další části práce se budou věnovat jen statické analýze PHP, je nutné se krátce zmínit o vlastnostech tohoto skriptovacího jazyka. Jedná je o interpretovaný jazyk, který vznikl v roce 1996 s původním jménem „Personál Home Page“. Následně prošel velkými změnami a dnes je již znám pod jménem „Hypertext Preprocesor“. PHP je především jazykem využívaný k dynamické webové prezentaci. Kód se zpracovává na serverové části a po zpracování se výsledek odesílá prohlížeči. PHP nemá žádný bod vstupu (jako funkce main v C programech). Interpretace začíná od začátku skriptu.

Protože byl jazyk vytvářen pro účel internetových prezentacích, je k nim ideálně přizpůsoben. Pokud programujeme menší aplikace je syntaxe je velmi jednoduchá a z velké části připomíná jazyk C nebo novější C++. Obsahuje spoustu vestavěných funkcí pro práci s řetězci, databází a tzv. „webovými proměnnými“ (URL řetězec a podobně).

V novějších verzích jazyka najdeme pokročilé techniky objektového programování. Dnes se již jazyk prakticky neliší od moderních neinterpretovaných jazyků s podporou pro objektové programování.

3.4.2 Úskalí analýzy jazyka PHP

Vyhledáváním v globální počítačové sítě brzo čtenář zjistí, že přestože existuje řada společností vyvíjející nástroje pro statickou analýzu, jen malé procento z nich je určeno pro jazyk PHP. K tomu je nutné doplnit, že řada z nich je úzce specializovaná na konkrétní účely (například hledání určitého typu chyb). Najít a uplatnit v praxi takový nástroj je s jedním z cílu této práce, proto je vhodné se zamyslet proč tomu tak je.

PHP je velmi dynamicky měnící se jazyk a s novými verzemi se mění i výsledná verze aplikací. Nástroj by tedy musel být dostatečně udržovaný, aby pružně reagoval na změny ve vývoji jazyka. Kromě změny syntaxe jsou přidávány nové funkce nebo třídy a mnoho z nich projde úpravou. Což jistě není tak velký problém pro širší komunitu nebo firmy s dostatkem financí.

Paralelně se dostáváme k dalšímu důvodu. Interpreti jazyka PHP jsou prakticky zdarma a jazyk je velmi otevřený. Na internetu má pověst „černého koně“. Ve výsledku je používán pro menší nebo nekomerční projekty. Samozřejmě se najde mnoho výjimek, ale v porovnání s platformou .NET nebo Java server pages (dále jen JSP), které jsou využívaný takřka jen pro profesionální komerční aplikace, zde rozdíl je. Důvodů je mnoho. Za jazykem zatím nestojí silné firmy, jako Sun nebo Microsoft. Nejsou k dispozici nejkvalitnější vývojová prostředí. Respektive jich je velmi mnoho, ale jen málo kvalitních. Statická analýza je vhodná pro větší projekty s vícečlennými týmy, kde má cenu sjednotit a analyzovat kód. Takových projektů má patrně daleko více komerčněji zaměřené JSP nebo .NET.

3.5 Dostupné nástroje

3.5.1 PHP Security scanner

PHP Security scanner je nástroj, který hledá slabiny ve zdrojovém kódu. Jeho zdroj je napsán v jazyce PHP, což je nespornou výhodou. Bohužel jeho kvality snižuje praktická absence dokumentace a omezená použitelnost.

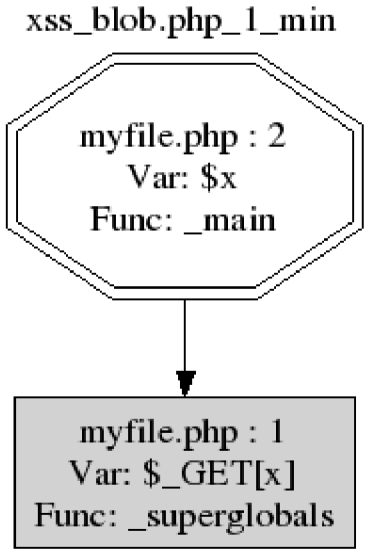
Po analýze zdrojového kódu nástroje zjistíme, že není příliš robustní a o rozložení kódu do syntaktického stromu si můžeme nechat jen zdát. Aplikace pracuje na základě prohledávání zdroje pomocí regulárních výrazů. Aktuální verze 1.0.2 obsahuje celkem 52 vzorů k problémům, které je schopné najít a nabídnout nápovědu k opravě. Nástroj neumožňuje definici nových metodických chyb a uživatel je odkázán na aktualizace. Jedná se zpravidla jen o použití vestavěných funkcí nebo superglobálních proměnných.

Nástroj je pro uvedené nevýhody v rámci této práce prakticky nepoužitelný a mám jisté pochybnosti i o použitelnosti v rámci jiných projektů.

3.5.2 Pixy

Další nástroj, který patří mezi statické analyzátory PHP kódu se jmenuje Pixy. Pixy se specializuje opět na slabiny zdrojového kódu. Konkrétně podává informace o tom, jak jsou skripty náchylné na útoky typu XSS a SQLi. XSS nebo-li „Cross-site scripting" je metoda narušení webových stránek využitím bezpečnostních chyb ve skriptech (jde především o neošetřené vstupy). Útočník je díky těmto chybám schopen do stránek podstrčit svůj JavaScriptový kód, což může vést ke trvalému nebo dočasnému vzhledu stránek a někdy i k odcizení citlivých údajů návštěvníků stránek nebo obcházení bezpečnostních prvků stránky. Naproti tomu SQLi nebo-li „SQL injekce" je metoda útoku, kdy je pomocí modifikace URL řetězce upraven SQL dotaz a v případě neošetření vstupů z adresy je takto možné odcizit citlivé údaje stránek, což často vede k úplnému převzetí kontroly nad stránkami.

Nástroj je napsán v jazyce Java a je solidně dokumentován. Pixy rozkládá zdrojový kód na syntaktický strom, ale není k dispozici možnost definovat nová pravidla. Vše je nastaveno jen na řešení výše zmíněných problému. Na domovských stránkách projektu je dokonce k dispozici webové rozhraní aplikace, které ukazuje práci aplikace a je možné takto otestovat své zdrojové soubory. Pixy jako výsledek analýzy ukáže zdrojový kód z vyznačenými řádky a informaci o chybě v zabezpečení. Přikládá velmi hezké grafy, kde ukazuje umístění chyby a její závislost v rámci skriptů. Ukázka výstupu tohoto nástroje je na následujícím obrázku č.7 a napravo (obrázek č.8) od něj je vstup, z něhož byl generován. Jak je vidět, vše vypadá velice přehledně.



Obrázek 7 : Výstup nástroje Pixy

```
$x = $_GET['x'];
echo $x;
echo $y;
```

Obrázek 8 : Příkladný vstup nástroje Pixy

Pixy je velmi kvalitní nástroj, i když s menším rozsahem možností použití. K účelům jenž byl koncipován patří jistě k nejlepším. Nevýhodou se může zdát to, že je psán v jazyce Java a ne PHP. Tuto drobnou vadu na kráse lze přehlédnout, protože zdrojové kódy nejsou volně k dispozici a nelze

předpokládat jejich modifikaci. V poslední verzi též nebylo možné zvalidovat celý projekt, ale byla k dispozici jen postupná validace po souborech. Nástroj se neustále vyvíjí a je možné očekávat pružnou reakci na případné změny v jazyku PHP. Výhody a možnost nasazení toho nástroje bude později prodiskutován s pracovní skupinou.

3.5.3 Php Sat

Php Sat je nástroj napsaný v jazyce C a je distribuovaný pro manažér balíčků Nix. Dokumentace nástroje je průměrná a věnuje se hlavně využití nástroje. Aplikace je určena k hledání metodických chyb v PHP aplikaci. Nástroj je ve vývoji, proto nová pravidla neustále přibývají. Nové chybové vzory nelze definovat, protože jsou implementovány a zkompilovány uvnitř jádra nástroje Chyba je v nástroji určena následující šesticí.

- **Code :** číslo nalezené chyby
- **Affected version :** verze PHP, které se chyba týká
- **Example :** příklad chyby, který nejvíce reprezentuje chybu
- **Usage :** situace kdy se uplatní tento chybový vzor
- **Why :** Vysvětlení chyby
- **Solution :** Vysvětlení, jak odstranit chybu

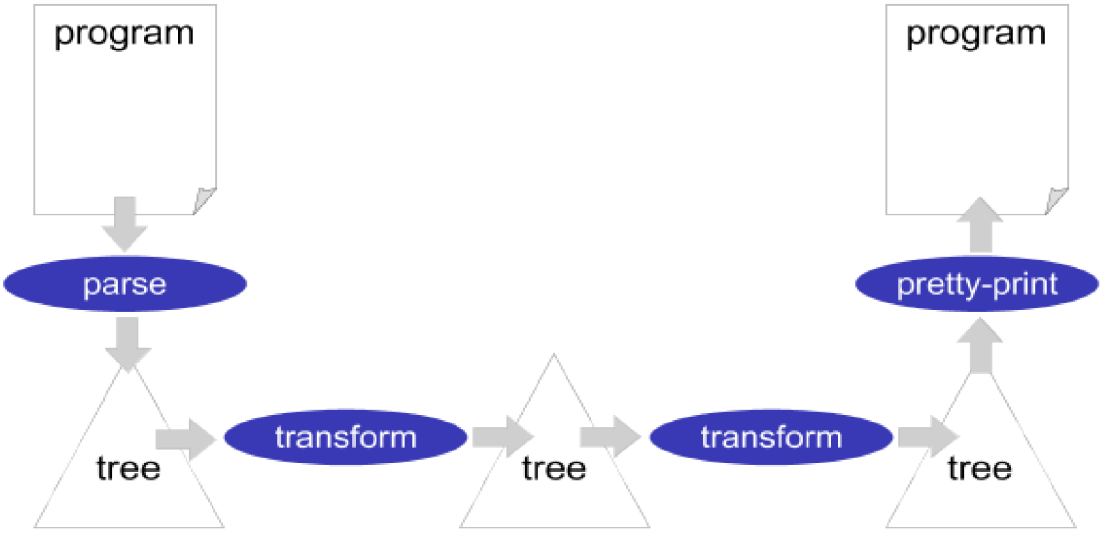
Chyby se pak následně dělí do pěti skupin Correctness, Exposing Info, Optimilization, Style a Malicious Code Vulnerability. Correctness jsou chyby v nesprávnosti kódu, jako špatný datový typ a podobně. Exposing Info je chyba, kdy může kód proniknout do okolního světa a umožnit tak lepší pozici útočníkům. Optimilization je chyba v kódu, která úpravou může přinést zlepšení výkonnosti skriptu. Style je chyba ve stylu zdrojového kódu, což může být značně subjektivní. Malicious Code Vulnerability typy chyb obsahují jen chybu, která může být škodlivá v rámci skriptu. Vzhledem k vývoji se rozčlenění může změnit.

V případě nalezení chyby je uživatel informován o řádku a kódu chyby. Nástroj umožňuje informace o nalezených chybových vzorech vkládat rovnou do zdrojového kódu PHP jako komentáře. Na závěr je třeba říci, že se jedná o velmi kvalitní nástroj. Bohužel díky malé nastavitelnosti se nástroj příliš nehodí pro nasazení do pracovní skupiny. Nemohl by odhalit všechny požadované vzory a neumožňuje reakci na nálezy.

3.5.4 Php Front

Php Front je aplikace od stejného autora jako Php Sat a je opět nástrojem pro statickou analýzu, ale má trochu jiný účel. Tento nástroj je primárně určen ke transformaci a generování PHP kódu. Je velmi podobný nástroji Java Front, od kterého ho má také podobné pojmenování.

V nástroji lze založit nový projekt a řadou pravidel provádět transformace. K těmto účelům využívá platformu Stratego/XT. Stratego je jazyk implementující softwarové transformace a XT je sada transformačních nástrojů. Transformaci zachycuje obrázek č.9. Je třeba doplnit, že cílový program nemusí být ve stejném jazyku. Jedná se tedy o velice mocný nástroj.



Obrázek 9 : Transformace v platformě Stratego/XT

Jak je již z úvodních vět patrné k definování transformace je nutná alespoň základní znalost gramatik a platformy Stratego. Jedná se o velice zajímavý nástroj, který by mohl být dále v práci použitelný. Umožňuje totiž transformace syntaktického stromu a tím prakticky reagovat na nalezené vzory. Použití bude dále prodiskutováno v pracovní skupině. Pro skupinu je mimo jiné důležité pozdější vytvoření transformací mimo rámec diplomové práce. Tomuto by muselo předcházet zaučení některého ze stálých členů týmu do jazyka Stratego. Tuto nevýhodu je nutné brát v potaz, i když je jazyk velmi intuitivní.

3.5.5 PHC

PHC je asi nejzajímavější z nástrojů., který překvapí velice rozsáhlou a dobře zpracovanou dokumentací. Jak již název napovídá, aplikace je napsána v jazyce C++. V posledních verzích se aplikace zredukovala na podporu vytváření modulů pro vyhledávání vzorů. Než je tedy možné započít analýzu konkrétního skriptu, je třeba vytvořit modul provádějící určitý úkon. V PHC se

pravidla vytváří pomocí objektového návrh. Podobně jako v nástroji PMD pro jazyk Java se dědí z třídy umožňující následné transformace syntaktického stromu.

Možnosti PHC nekončí u transformací a vyhledávání vzorů. Dokáží převést PHP syntaxi do XML reprezentace a zpět. Další velmi zajímavou volbou je export stromu do grafického formátu DOT. Již podle výše zmíněného stručného popisu je jasné, že nástroj velmi zapadá do předběžných požadavků pracovní skupiny. Menší nevýhodou je, že implementace musí probíhat v jazyce C++, ale jak už již bylo psáno PHP je syntakticky velice podobné tomuto jazyků. Nástroj bude později skupině prezentován.

3.5.6 Vlastní nástroj

Další možnou volbou je návrh a implementace vlastního nástroje. Bezespornou výhodou by byla implementace namíru. Obrovskou nevýhodou následně vynaložený čas na vývoj takového nástroje, který by bylo vhodnější věnovat výzkumu požadavku a postupné integrace nástroje. Tuto možnost si proto dovolím zamítnout.

Daleko lépe vyznívá využití stávajícího nástroje a následné rozšíření, tak abychom odstranili jeho nevýhody a implementovali jen nové požadované vlastnosti. Pro tento typ implementace se z výše popsaných nástrojů hodí jen ty, které poskytují jen hrubou pracovní kostru v podobně nástrojů (PHC a PHP Front).

3.6 Vyhodnocení a výběr nástroje

Ve výsledku vychází PHC jako nejlepší volba. Všechny klady tohoto nástroje již byly zmíněny v předchozích kapitolách. Je nutné se zamyslet nad integrací tohoto nástroje do reálného provozu v přiřazeném vývojovém týmu. PHC je jen sada knihoven a aplikuje se vždy jen na jednotlivé soubory. Některé jednoduché kontroly by také byly pomocí tohoto nástroje zbytečně zdlouhavé a implementačně náročné.

Všechny tyto klady a zápory byly předneseny kontaktnímu pracovníku týmu pro vývoj informačního systému VUT Ing. Michalu Juroszovi. Zde klady nástroje narážely na nedostatečnou znalost jazyka C++ u většiny programátorů a pozdější rozšiřitelnost v rámci vývojového týmu byla jedna z hlavních bodů požadavků tohoto týmu. Na konci rozhovorů se dospělo, jak už většinou bývá zvykem, ke kompromisu. Bude implementován nový nástroj, který bude splňovat všechny požadavky, a aby se předešlo větší obtížnosti implementace a tím pádem časové náročnosti bude implementována jen jakási kostra schopna spouštět a zpracovávat pluginy různých jazyků nebo dokonce nástrojů. Je jasné, že takový nástroj bude muset mít dostatečně velkou možnost konfigurace. Mezi počáteční, plně podporované pluginy byli odsouhlaseny pluginy typu PHC a klasického PHP pro jednodušší úkoly obsahující regulární výrazy nebo prohledávání pole tokenů po lexikální analýze.

Protože bude PHC jedním s typů pluginů výsledné aplikace, bude nezbytné se podrobně seznámit s jeho implementací a rozhraním dříve než k dojde k samotnému návrhu a následné implementaci. Ve výsledku bude tedy knihovně PHC v rámci této práce věnována samostatná kapitola, kde se čtenář doví více o tom jak tato velice zajímavá knihovna funguje.

4 PHP Compiler

4.1 Úvodem

Z předchozích odstavců může být čtenář zmaten zda se jedná o knihovnu, samostatný nástroj nebo sadu pluginů. V komunitě okolo tohoto nástroje tímto vládne také lehký zmatek, proto je nutné si tuto podstatnou věc ujasnit. PHP Compiler nebo-li PHC je podle definice samotných autorů framework, pro statickou analýzu PHP kódu, transformaci a kompilaci. V současné době je ve vývoji, a protože vývoj už trvá nějaký ten rok je nutné předpokládat, že hned tak brzy ukončen nebude. V aktuální verzi tedy nemůžeme počítat s kompilací, což vzhledem k tématu této práce a k němu korespondující aplikaci nikterak nevadí. Statická analýza a transformace je již na velmi dobré úrovni a existuje jen několik málo výjimek, s kterými si PHC neporadí.

Framework je založen na pluginech, které provádějí akce se syntaktickým stromem zdrojového kódu. Veškeré akce jsou implementovány pomocí nástrojů dědičnosti, kterým disponuje jazyk C++. Plugin se před prvním spuštěním musí zkompilovat. Pro jednoduší kompilaci má nástroj definován shellový skript.

Ke svému běhu nástroj vyžaduje prostředí unixového typu, překladač GCC ve verzi 3.4.0 nebo vyšší. Pokud uživatel bude chtít manipulovat skrze PHC s formátem XML, je nutné mít nainstalovanou knihovnu Xerces-C++ pro parsování XML dokumentů. Protože PHC zvládne i výstup v podobě grafu ve formátu DOT, je pro jeho prohlížení potřeba mít vhodnou aplikaci (např. Graphviz).

Framework je možné stáhnout na domovských stránkách tohoto projektu. Jak již bývá v aplikacích pro uniové operační systémy zvykem, k dispozici je zdrojový kód s make souborem pro překlad celého projektu.

4.2 PHC pod lupou

4.2.1 Gramatika v nástroji

Nástroj, který pracuje s nějakým jazykem a poskytuje takové sofistikované akce jako právě PHC, potřebuje mít ve vhodném formátu definovanou gramatiku jazyka, se kterou operuje. Gramatika, jako soubor pravidel, se dá reprezentovat mnoha formáty. PHC plně využívá pro definici formát maketea. Soubor s pravidly v tomto formátu je možné najít v jednom ze souborů nástroje.

Tomuto formátu a jak z něj PHC generuje C++ kód, jenž umožňuje ve výsledku velmi pohodlné procházení PHP kódem, je věnována následující kapitola.

4.2.2 Maketea

Styl gramatického formalismu, jaký je používán v maketea je někdy nazýván jako „objektově orientovaná“ bezkontextová gramatika. Důvodem je, s jakou lehkostí je formát možné převést na dostatečně popisující C++ kód.

Ve formátu je možné nalézt tři druhy symbolů. Terminální, non-terminální symboly a značky. Non-terminální symboly jsou vyjádřeny v Backus-Naurově formě. Ve zkratce jde o formu pro zápis formálních jazyků od autorů John Backuse a Peter Naura.. Detaily této formy jsou mimo rámec této práce. V gramatice v PHC se oproti tomu, jak je ve formálních jazycích zvyklé, píší non-terminály malými písmeny. Značky jsou označovány uvozovkami (např. Značka „abstract“).

V následném C++ kódu má non-terminální symbol vždy korespondující třídu s předponou „AST_“. Pokud tedy máme non-terminál „a“, korespondující třída má název „AST_a“. Terminální symboly jsou naopak reprezentovány třídou s předponou „TOKEN_“. Konstrukce názvu je obdobná jako u non-terminálů. Každá třída typu token obsahuje vlastnost s názvem „value“. Typ tohoto atributu třídy závisí na gramatice a v naprosté většině se jedná o typ řetězec. V jiném případě je typ definován v hranatých závorkách přímo v gramatice. Pokud je takto explicitně definován typ, dostane metoda atribut s názvem „source_rep“, který je vždy typu řetězec. V této vlastnosti je uchována hodnota zdrojového tokenu. Na druhou stranu značky nejsou ve výsledném objektovém kódu reprezentovány žádnou třídou. Mohou však být reprezentovány atributem . Pravidlo, které bude mít ve výsledném C++ kódu atribut „is_foo“ v třídě „AST_a“ vypadá takto:

a ::= ... "foo"? ...

Z předcházejícího pravidla je vidět, že řetězec „::=“ odděluje pravou a levou stranu přepisovacího pravidla. Na levé straně leží non-terminal a na pravé straně kombinace terminálu, non-terminálu a značek. Pravidla jsou v maketea kategorizovány a existují dva základní druhy. První z nich je pravidlo v následujícím tvaru :

a ::= b | c | ... | z

V pravidle v tomto tvaru jsou symboly „a“, „b“, „c“ až „z“ non-terminály. Symbol vertikálního oddělovače se používá k vyjádření volby pravé strany. Tato konvence je v oboru formálních jazyků velmi často používána. Jak je z předchozích odstavců patrné, ve výsledku jsou symboly reprezentovány třídami a to tak, že třídy symbolů na pravé straně dědí ze symbolu na levé straně. V tomto příkladě třídy korespondující se symboly „b“ až „z“ dědí z třídy korespondující k symbolu „a“. Pokud existuje více pravidel, kde tentýž symbol figuruje několikrát na pravé straně,

potom dědí vždy ze všech tříd. Na následujícím kódu by třída AST_c dědila ze dvou tříd korespondujících k symbolům na levé straně.

a ::= c | ..., b ::= c | ...

Již na předchozím příkladě je vidět výhoda použití objektové reprezentace v jazyku C++. Jazyk má velice pokročilé metody pro implementaci objektového návrhu a jako jeden z mála podporuje přímou vícenásobnou dědičnost a ne jen nepřímou skrze rozhraní.

Druhým základním pravidlem jsou pravidle tohoto typu :

a ::= b c ... z

V kódu jsou „b“, „c“ až „z“ volitelné non-terminální, terminální symboly nebo značky. V tomto pravidlu jsou zakázány jakékoliv disjunktní operátory vertikální čárky, ale jsou povoleny tři unární operátory „*“, „+“, „?“, které se píši ihned za operand, tedy symbol. Jejích význam je totožný se stejně značenými symboly v regulárních výrazech a označují četnost nebo volitelnost. Pokud je symbol iterován ve výsledném kódu, je označena třída prefixem „AST_list_“, která bude dědit ze třídy označené řetězcem „STL_LIST“. Dále se před symboly může vyskytnout znak dvojtečky. Řetězec před dvojtečkou nemá v rámci pravidla gramatiky žádný vliv, ale má vliv na prosté pojmenování a je podle něj pojmenována výsledná korespondující třída. To je velmi výhodné a dovoluje tak z gramatiky generovat přehlednou objektovou strukturu.

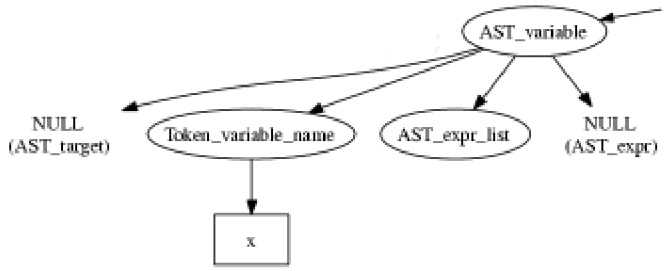
Pro příklad je na následujícím kódu možné vidět, jak je generována ze dvou příkladných pravidel.

```
expr ::= ... | variable | ... ;
variable ::= target? variable_name array_indices:expr?* string_index:expr? ;

class AST_variable : virtual public AST_expr
{
public:
  AST_target* target;
  AST_variable_name* variable_name;
  AST_expr_list* array_indices;
  AST_expr* string_index;
}
```

Z předchozích odstavců je patrné, že každá třída korespondující se symbolem na levé straně přepisovacího pravidla druhého typu, obsahuje ke každému korespondujícímu symbolu na pravé stejně pojmenovaný atribut. Je třeba doplnit, že pokud nejsou názvy explicitně definovány, volí se podle názvu atributu a v případě iterace obsahuji příponu „s“. Stejně jako v reálném anglickém jazyce, když je třeba vyjádřit množné číslo běžného slova. Dále může čtenáře zarazit kombinace symbolů „?“ a „*“. Konstrukce (a*)? značí volitelný seznam symbolů „a“. Naopak konstrukce (a?)* reprezentuje seznam volitelných symbolů „a“. Pokud je seznam definovaný, ale neobsahuje žádné prvky bude jednoduše prázdný. Pokud bude neúplně definován bude mít hodnotu „NULL“.

Ve výsledku může být jeden řádek PHP kódu „$x;“ reprezentující proměnnou, reprezentován objektovou hierarchií, jak ji ukazuje obrázek č.10.



Obrázek 10 : Grafová reprezentace PHP kódu v PHC

V následující kapitole bude dokončen popis reprezentace syntaktického stromu objektovým modelem a to třídami „Tree Visitor“ a „Tree Trasform“, které slouží přímo pro procházení a transformaci stromu.

4.2.3 Aplikační rozhraní

V předchozí kapitole byli třídy rozčleněny na dva základní typy, které se liší podle předpony „AST_“ nebo „TOKEN_“. Dále budeme tyto třídy v této práci nazývat jednoduše AST třídy případně TOKEN třídy.

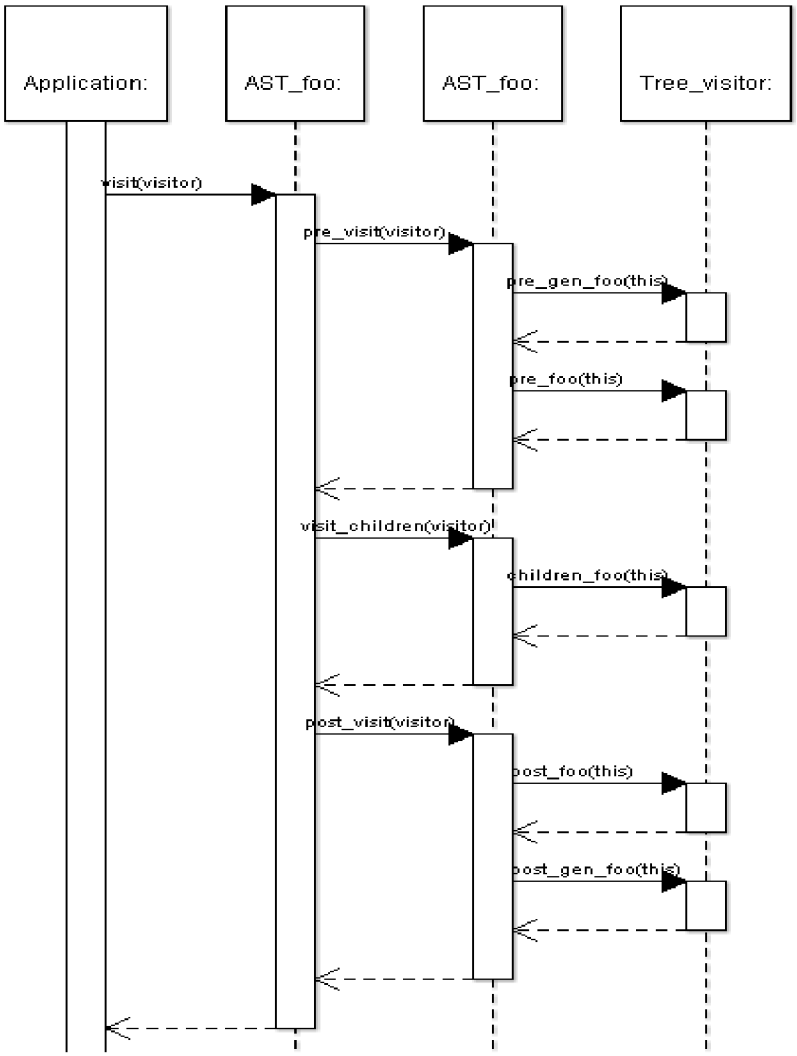
TOKEN třídy obsahují kromě atributu „value“, jenž byl zmíněn v předchozí kapitole, také metodu „get_value_as_string“, která vrací hodnotu ve tvaru řetězce. Pokud obsahuje atribut „source_rep“ obsahuje i metodu „get_source_rep“.

AST třídy obsahují metody pro hloubkové porovnávání, porovnávání vzoru, klonování, volání „Tree Visitoru“ a „Tree Transform“ tříd. Každé metodě bude věnovaný samostatný odstavec.

Hloubková ekvivalence je implementována metodou „deep_equals(Object* other)“, která prochází celý objektově reprezentovaný strom a dokáže zjistit shodu komentářů nebo čísla řádku a podobně.

Klonování je implementováno metodou „deep_clone“. Hloubkové klonování udělá kopii daného stromu. Všechny ukazatele v novém stromu budou nově přiřazeny, takže nebudou korespondovat se starým stromem. Jedinou výjimkou je, když se klonuje takzvaný WILDCARD objekt. V tomto případě se vrací přímo ten samý objekt.

Porovnávání je implementováno metodou „match(Object* pattern)“. Porovnání, na rozdíl od hloubkové ekvivalence, nedokáže porovnávat řádky zdrojového kódu a komentáře. Zvládá, ale navíc manipulaci s takzvanými WILDCARD objekty. WILDCARD není speciální třída, ale je to instance, která je definovaná v jednom z hlavičkových souborů. Pokud metoda „match“ obdrží referenci na objekt WILDCARD, porovnávání vlastností se přeskočí a hodnoty v porovnávaném objektu „patter“ se nahradí hodnotami ze stromu.

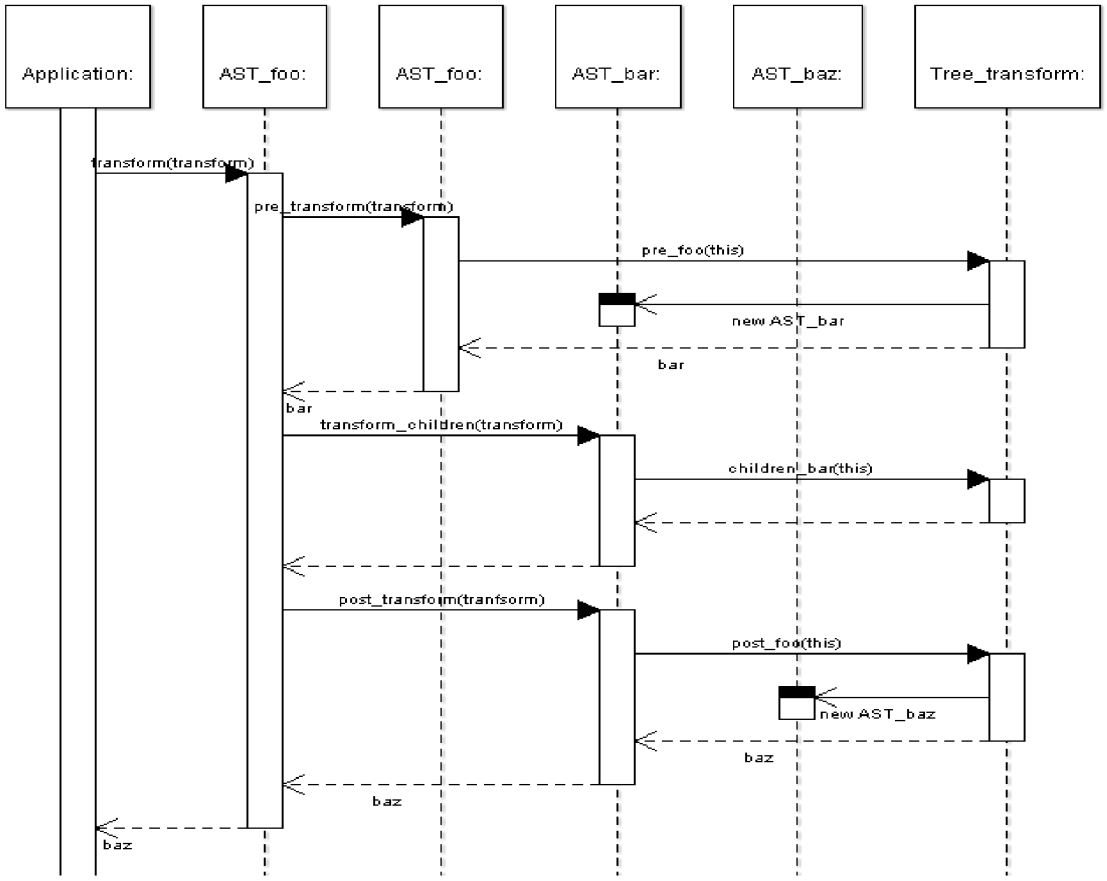


Obrázek 11 : Aplikační rozhraní „Tree Visitor“

Aplikační rozhraní třídy „Tree visitor“ poskytuje tyto metody : „visit(Tree_visitor*)“, „pre_visit(Tree_visitor*)“, „visit_children(Tree_visitor*)“ a „post_visit(Tree_visitor*)“.

Metoda nazvaná „visit“ volá zbylé tři metody v pořadí, v jakém byly napsány v předchozí větě. Funkce dalších metod je lépe ukázat na příkladě. Máme uzly N1, N2, N3, kde N1 dědí z N2 a N2 z N3. Metoda „previsit“ volá nejdříve metody „pre_ N3“, následně „pre_N2“ a tak dále. Pokud se vyskytuje vícenásobné dědění, metoda nedělá výjimku a volá všechny metody předcházejících uzlů od nejvzdálenějšího rodiče. Metoda „visit_chidren“ naopak volá postupně všechny potomky uzlu. Konečně metoda „post_visit“ volá uzly v opačném pořadí než „pre_visit“ a to „pre_N1“, „pre_N2“ a tak dále. Visuální ukázka se nachází na obrázku č.11 v podobě sekvenčního diagramu.

Aplikační rozhraní třídy „Tree Transformu“ zde popisováno nebude. Důvodem je, že nástroj nebude využit k refaktorizaci kódu. Opravy jsou, jak bude zmíněno více v závěru této podkapitoly, tímto nástrojem ne zcela vhodné. Pro zajímavost je přiložen obrázek č.12, opět v podobě sekvenčního diagramu.



Obrázek 12 : Aplikační rozhraní „Tree Transform“

4.3 Práce s frameworkem

4.3.1 Vstupy

Vstupním bodem frameworku je spustitelný soubor phc, který v závislosti na parametrech spouští pluginy nebo provádí jiné akce. Soubor phc je čistě shellovská aplikace a postrádá tedy jakékoliv grafické uživatelské rozhraní. Na následujících řádcích je kompletní výpis možných parametrů k aktuální verzi 0.1.6 :

phc **[VOLBA]** ... [SOUBOURY] ...

-h, --help	Vytiskne krátkou nápovědu a ukončí se.
--full-help	Vytiskne delší nápovědu spolu se skrytými volbami a ukončí se.
-V, --version	Vytiskne číslo verze a ukončí se.
--dump-php	Vstupem slouží php soubor, výstupem je pak v případě validního kódu tentýž kód zformátovaný dle PHC . Implicitně vypnuto.
--dump-ast-dot	Výstupem je abstraktní syntaktický strom v DOT formátu. Implicitně vypnuto.
--dump-ast-xml	Výstupem je abstraktní syntaktický strom v XML formátu. Implicitně vypnuto.
--compile-time-includes	Když je možnost PHC nahradí include příkazy v PHP kódu přímo samotným parsovaným kódem. Implicitně vypnuto.
--tab=STRING	Řetězec, který je vkládán jako odsazení úrovně ve formátovaném výsledném PHP kódu.
--run	Jako první soubor slouží PHC plugin, který bude spuštěn Druhým souborem je zdrojový kód PHP.

Dalším vstupem, kromě konfiguračních parametrů, je zdrojový kód v jazyce PHP a při správně zvoleném parametru také zkompilovaný plugin ve vhodném formátu (viz předchozí kapitola).

4.3.2 Kompilace pluginů

Kompilace probíhá, jak už bylo zmíněno v úvodu, malým shellovým skriptem. Ten volá překladač g++ se spoustou různých voleb. Ve výsledku se tedy kompilace testovacího pluginu s názvem „helloworld“ provede velmi jednoduchým příkazem :

~/plugins_dir$ phc_compile_plugin helloworld.cpp -o helloworld.so

Jen je třeba doplnit, že se v PHC frameworku se pluginy pojmenovávají standartní příponou „.so“. Po zbytek práce a i v rámci souvisejícího aplikace to budeme považovat za pravidlo.

4.3.3 Výstupy

Výstupem z PHC může být buď samotný transformovaný, případně čistý kód. Obrazová podoba syntaktického stromu v DOT formátu nebo jeho XML reprezentace.

Pluginy mohou mít samostatné výstupy a jejich hodnota a formát záleží čistě na implementaci pluginu. Právě tato vlastnost se zdá největší výhodou, protože takto může být řešena zpětná komunikace pluginu s programy třetích stran.

4.3.4 S čím si framework neporadí

Jak již bylo na začátku této kapitoly zmíněno, PHC je stále ve vývoji a rozhodně si v PHP kódu neporadí se všemi záludnostmi. Protože bude PHC využíváno v korespondující aplikaci, je nutné se s těmito nedostatky blíže seznámit.

První z věcí, které zatím PHC nepodporuje, jsou návratové hodnoty z vložených skriptů metodami include a require. Při volání vkládání si PHC poradí jen se znakovými řetězci a ne s proměnnými a podobně. Dále negarantuje správné jedinečné vkládání metodami include_once a require_once. Také nepracuje úplně přesně s metodou get_included_files.

Jak je z výčtu vidět, prakticky všechny nevýhody se týkají vkládání souborů, což by vadilo při kompilaci PHP, ale ne příliš při jeho statické analýze.

4.4 Závěrem

Obrovskou nevýhodou, která plyne z principu, jakým PHC pracuje je, že PHC neudržuje počet mezer a vůbec vzhled kódu. Nastavení formátu výsledného kódu je sice možné, ale v jen velmi omezené míře.. Tímto se tento mocný nástroj umožňující refaktorizaci kódu degraduje pouze na nástroj pro statickou analýzu. Vzhledem k tématu této práce tento fakt není příliš na škodu. Rozvoji korespondující nástroje to může později, jak se říká přistřihnout křídla.

Implementačně zbývá tuto nevýhodu upřesnit a říci, že bude jako rozhraní použita jen třída „Tree Visitor“.

5 Požadavky a analýza

5.1 Úvodem

Nejdříve se je potřeba zamyslet nad požadavky na korespondující nástroj. Prakticky všechny plynou z teoretických částí před touto kapitolou, je ale důležité je sepsat jako celek a analyzovat. Další požadavky byly získány formou rozhovoru s kontaktní osobou programátorského týmu VUT.

Bylo by vhodné také pojmenovat tento korespondující produkt, ke kterému se váže zbytek této práce. Je možné pojmenovávat onen projekt dílem, aplikací či podobně, ale každý produkt by měl mít své jméno. Produkt byl pojmenován jednoduše podle svého účelu Ve zkratce kontroluje metodiku programování. Název tedy zní Metodika.

5.2 Analýza

První a pro projekt zásadní požadavek byl ten, že by měl být velice snadno rozšiřitelný. Tento fakt vede ke psaní pluginů, tedy samostatných části, které lze libovolně přidávat a odebírat. Zmíněný požadavek jde ruku v ruce s dalším, pro klasické aplikace ojedinělým, požadavkem. Metodika by měla umět pracovat s pluginy různých jazyků. Nutná je především podpora jazyka PHP, který všichni programátoři v pracovní skupině musí ovládat. Pluginy frameworku PHC jsou psány jazykem C++. Podpora by měla být tedy i pro ně. Aplikace Metodika by měla být také dostatečně flexibilní v rámci různých nastavení. Pluginy by mělo jít jednoduše zapínat a vypínat.

Kromě klasického standardního výstupu by si Metodika měla poradit i odesíláním informačních emailů. Vzhledem k charakteru aplikace je zajímavý i způsob integrace, který pravděpodobně bude opět nestandardní. Metodika se nainstaluje na server, kde běží program na správu verzí Subversion. Při „commitu“, tedy potvrzení nového souboru, se Metodika spustí.a provede definovanou akci.

Mělo by se tedy jednat dávkový program bez grafického uživatelského rozhraní. Prostřednictvím konfiguračních souborů a příkazové řádky by mělo docházet k ovládání a nastavení. SVN ve vývojovém týmu běží na operačním systému LINUX. Metodika by teda měla být schopna provozu pod systémy unixového typu. Pluginy by měly být schopny být aplikovány na každý soubor projektu nebo na čistý text ze standardního vstupu. Požadavky na výstup byly formulovány velice obecně. Ve výsledku jde hlavně o to, aby se plugin, který je primárně určen k hledání vzorů v kódu, staral o výstup samostatně. Výhodou bude grafická reprezentace výsledků ve formě grafů či jednoduchého HTML výstupu. Každý plugin by tedy měl podat informaci o důvodu chyby, možná řešení chyby a číslo řádku, na kterém byl chybový vzor vyhledán.

V posledních fázích rozhovoru se společným úvahami dospělo k závěru, že by bylo vhodné chybové vzory nejen vypisovat, ale i je někde mít alespoň trochu formálně napsané. Tedy půjde o nějaký dokument se soupisem všech aplikovaných pravidel vyhledání vzorů v rámci projektu. Aplikace Metodika by mohla na tento dokument a na jeho konkrétní části odkazovat a programátorům pomoci lépe pochopit, co udělali špatně. Pro tento účel se výborně hodí webové stánky. Pro jednoduché použití a přehledné úpravy se výborně hodí systém Wiki. Pokud by měl někdy projekt překročit hranice svého primárního určení a byl tak dostupný pro více programátorských týmů, bylo by vhodné pro něj zajistit samostatnou doménu vyššího řádu a případně svůj vlastní publikační systém, protože systém Wiki by už nemusel stačit.

5.3 Další požadavky seznamem

K rozhovorům byl zhotoven seznam požadavků a každá položka je v této práci vedena jako jedna podkapitola. Jedná se o seznam metodik a možných chyb, které by v kódu stálo za to vyhledávat. Seznam nevznikl ze dne den, ale byl výsledkem delších rozhovorů a neposledně zkušeností ze stran konzultanta Ing. Michala Jurosze. Kromě jeho, byly uplatněny i zkušenosti jiných osob, které prostřednictví globální internetové sítě svoje zkušenosti v této oblasti zpřístupnily. Po sepsání kompletního seznamu byly krok po kroku ještě jednou všechny body zkonzultovány a případně upraveny. V následující kapitolách je tedy tento list zdokumentován.

5.3.1 Vynechání tabulátoru

Tabulátor použitý programátory pro odsazování může být problém. V některých systémech je reprezentován jiným počtem odsazovacích mezer, a proto může dojít k nechtěnému rozhození programového kódu. Je tedy třeba v případě použití tabulátoru pro odsazení programátory upozornit.

5.3.2 Detekce výstupu

V dobách, kdy je velmi populární architektura MVC (model – view – controller) není k podivu, že jsou některé soubory, moduly určeny k jiným účelům než výstupu. Dokonce je výstup v takových souborech nežádoucí a porušuje pravidla architektury. Některý z pluginů by tedy měl kontrolovat, zda některé ze souborů negenerují výstup.

5.3.3 Detekce testovacího výstupu

PHP podporuje řadu funkcí pro „debugovací“ a testovací výpisy. Žádná z nich však nemá v publikovaném kódu co dělat. Jedná se většinou o nepozornost, ale i tu je možné kontrolovat. Jde o funkce „print_r“, „var_dump“ a podobně.

5.3.4 Výstup jen pomocí jednoho konstruktu

PHP má pro výstup vestavěný několik různých funkcí a dva hlavní jazykové konstrukty echo a print. Konstrukty zdánlivě vypadají jako funkce, ale parametr je předáván v uvozovkách. Aby bylo vše ještě trochu zamotanější, umožňují některé konstrukty předávání parametrů i v závorkách jako funkce.

Hlavní rozdíl mezi konstrukty echo a print je v tom, že print se chová daleko více jako funkce a dokonce i vrací hodnotu. Ve výsledku jde dosáhnout pomocí obou funkcí stejné výsledky a využití výhod každé z nich příliš práce neušetří. Bylo by tedy vhodné v případě, že bude výstup v daném souboru povolen, povolit jen jeden z konstruktů a říci jestli jej používat se závorkami či uvozovkami.

Po rozhovorech a z důvodu větší rozšířenosti konstruktu echo v současném stavu projektu byl jako povolený konstrukt pro výstup zvolen echo s uvozovkami.

5.3.5 Kontrola PHP tagů

PHP není žádným nováčkem na poli skriptovacích jazyků a v současné době již existuje ve verzi 5. Postupně se v PHP objevilo několik otevíracích tagů <?php, <?, <%, <?=, <%= a také dva uzavírající tagy ?> a %>. Zde je již vidět nevýhoda v podobě nepřehlednosti, který byla teoreticky rozebrána v prvních kapitolách této práce.

Bohužel tento problém přebíhá i do úrovně zabezpečení. Některé přísně nastavené interprety jazyka PHP akceptují pouze dvojici <?php, ?>. V případě jiných otevíracích PHP tagů považují skript za zcela standardní text a vše posílají na stranu klienta. Nic netušící uživatel má celý zdrojový kód jako na podnose. Tento fakt není vhodné brát na lehkou váhu a akceptovat jen dvojici <?php, ?>

5.3.6 Maximální délka řádku

Tento problém byl také rozebírán v teoretické části této práce, a proto je jen třeba pevně stanovit onu hodnotu, na kterou bude omezena maximální délka řádku. Po krátkém rozhovoru byla maximální délka řádku stanovena na 130 znaků. Tuto hodnotu nebude problém, kdykoliv jednoduše změnit.

5.3.7 Povinný „return“ ve funkcích

Již začínající programátoři ví, že rozdíl mezi funkcí a procedurou je takový, že funkce vrací hodnotu. PHP má syntaxi velice podobnou jazyku C a stejně tak definuje jen pojem funkce. Procedura je tedy funkce, která nevrací nic, tedy vrací právě „nic“. Ve výsledku by tedy měla mít každá funkce příkaz return, i když nevrací nic (v PHP implementováno jako prázdný return).

5.3.8 Složené závorky na novém řádku

Pro přehlednost bývá v některých pracovních týmech domluva, že každá složená závorka ohraničující nějaký blok příkazů by měla být na samostatném řádku. Konzultant pracovního týmu je stejného názoru, proto tento požadavek figuruje také v tomto seznamu.

5.3.9 Komentovaný pád „case“

PHP podporuje klasickou konstrukci switch, case, break, default. O významu konstrukce se tato práce šířit nebude a jistě je všem dobře známa. Musíme se však zmínit o malém problému v přehlednosti, který může nastat v případě vynechání příkazu break. Průchod programem potom propadne směrem dolů dokud nenarazí na „break“. To může vést k zmatení, a proto je doporučováno vynechání breaku komentovat. Aplikace Metodika by měla být schopná tento prohřešek v přehlednosti vyhledat a programátora na něj upozornit.

5.3.10 Indexace asociativních polí

V indexaci asociativních polí, kde je klíčem řetězec může také dojít k nejednoznačnosti plynoucí z možnosti jazyka PHP. V jazyku je možné řetězové konstanty zapisovat jak v jednoduchých uvozovkách tak i ve dvojitých. Díky tomuto faktu může být jednoduší zapisovat složitější řetězce postupným střídáním různých uvozovek. Ve výsledku se tato „vymoženost“ přenáší i na indexace pole, kde každý programátor preferuje jiný druh uvozovek. PHP dokonce umožňuje vkládat libovolný počet mezer mezi operátor hranaté závorky a uvozovky. Po konzultaci bude metodika za účelem přehlednosti dovolovat jen jednoduché apostrofy bez jakýchkoliv mezer.

Další z věcí, která se u programátorů PHP stává je, že v řetězcích pro indexace polí používají nepovolené znaky. Například znak „-“ v takovém řetězci způsobí varování interpreta PHP. Je tedy nutné brát tento fakt jako požadavek pro kontrolu.

5.3.11 Definice pole s poslední čárkou

Pole se v PHP inicializuje konstruktorem array, který obsahuje jako seznam parametrů hodnoty indexů pole. Aby se předešlo podobnosti s funkcí, mnoho programátorů používá jeden oddělovač čárky na konci navíc. Dalším důvodem je možné předcházení syntaktické chyby v případě přidání nového prvku. Denice pole o třech prvcích tedy vypadá následovně : array(1,2,3,). Já osobně například tento programátorský návyk nemám, ale vidím v něm jisté výhody. Po konzultaci patří i tento bod mezi požadavky pro aplikaci Metodika.

5.3.12 Mezera mezi klíčovými slovy if a while

Mezera mezi klíčovým slovem if a příslušným výrazem v závorkách bývá někdy u programátorů k vidění a má své opodstatnění. Lze takovým elegantním odlišit klíčová slova od funkcí. Metodika má pomoci tento zvyk prosadit do pracovního týmu.

5.3.13 Mezera mezi operátory

Jestli předchozí požadavek byl poměrně unikátní, tento patří mezi jasné znaky dobrého programátorské vychování. Metodika bude tedy kontrolovat, jestli se mezera nachází za a zároveň před každým operátorem, který programovací jazyk PHP nabízí.

5.3.14 Komentování vnořených závorek

V programování obecně se často vyskytují složené závorky v velkém sledu za sebou. Programátor, který zrovna s kódem nepracoval je následně zmaten, která uzavírající závorka patří k jaké konstrukci. Mnoho programátorů pro větší přehlednost proto komentuje uzavírací závorky. Aplikace metodika by měla umět najít větší počet uzavíracích závorek bez komentáře. Případně také přihlédnout na délku kódu mezi jednotlivými závorkami. Ten může situaci ještě více znepřehlednit.

5.3.15 Používání otazníkové konstrukce

PHP jako i mnoho dalších programovacích jazyků podporuje konstrukci otazníku a dvojtečky, která je přehlednější než konstrukce „if“ a „else“ v případě, že se jedná v obou výrazech o jeden příkaz. Bylo by vhodné, kdyby program Metodika dovedl upozornit na to, že bylo vhodnější použít otazníkovou notaci místo klasického „else if“.

5.3.16 Vynucené používání složených závorek

Jak již bylo zmíněno v programování neznamená vždy, co je kompaktnější je také přehlednější. Když použijeme složené závorky u každé konstrukce „if“ a „else“, dosáhneme tak větší čitelnosti a můžeme předejít některým chybám, které pramení z návyků nepoužívat složené závorky u jednoho příkazu v konstrukci „if“ a „else“.

5.3.17 Prefix u tříd

Mnoho programátorů používá pro lepší rozeznatelnost názvu tříd prefix. Většinou se jedná o prefix „C“ nebo „T“ podle hlavního jazyku projektu. V pracovním týmu a konkrétně i v hlavní projektu, pro který bude software vyvíjen, ale v podstatě žádný větší počet tříd neexistuje. Proto tento požadavek nemá velkou váhu a jedná se jen o jakousi doplňkovou vlastnost, která by z implementačního hlediska nemusela být příliš obtížná.

5.3.18 Prefixy u vybraných proměnných

Stejně jako minulý požadavek není ani tento zatěžkán velkou prioritou a jedná se spíše jen o nápad do budoucna. V jednoduchosti jde o to, že pro pojmenování pravdivostních proměnných se často vybírá prefix „is_“ nebo u privátních proměnných třídy předpona „m_“. Po rozhovorech bylo jasné, že ohledně tohoto požadavku panují na straně konzultanta rozporuplné reakce a s implementací se může počkat na pozdější dobu, kdy již nástroj bude integrován.

5.3.19 Kontrola vkládání ze superglobalnich proměnných

V PHP je velice rozšířeno modulární programování. Pro jeho podporu zde existují funkce pro vkládání souborů a existuje zde reálná možnost přepsání adresy externě z klientské strany prohlížeče. Jde zejména o takzvaný „Cross-site scripting“ nebo je dokonce v případě špatného nastavení interpretu možné vložit soubor s kódem, který je na jiném serveru. Útočník následně může spustit svůj škodlivý kód. Více se tomuto tématu věnovala teoretická část v úvodu této práce.

5.3.20 Seznam povolených globálních proměnných

Globální proměnné bývají velice dobrými pomocníky pro přístup do vzdáleného kódu. Existují však jiné, čistější formy programování, přesto není v podstatě menší, předem dohodnutý počet globálních proměnných na škodu. Proto se od nástroje očekává, že by mohl vyhledat prohřešky proti stávajícímu seznamu globálních proměnných. Tedy jinak řečeno, jestli si programátor nezavádí nějakou svou novou globální proměnnou.

5.3.21 Konstrukce else if na jednom řádku

PHP je vzhledem k syntaxi velmi benevolentní jazyk. Konstrukce „else if“ je v PHP brána jako dva příkazy a ne jako v některých jiných jako jediná konstrukce. Z toho vyplývá možnost udělat mezi těmito klíčovými slovy prakticky jakýkoliv počet mezer i nových řádku. K celkové přehlednosti kódu přispívá, když jsou else a if oddělené jednou mezerou a jsou na stejném řádku.

5.3.22 Dočasné vypnutí detekce chyb

V teoretické části v jsem se zmínil o současných možnostech již existujících statických analyzátorů pro jiné jazyky. Většina z nich umožňuje programové vypnutí jejich činností pomocí speciálního komentáře vložené přímo do kódu . Tato možnost by byla určitě velmi výhodná i pro pracovní skupinu této práce, tedy vývojový tým informačního systému VUT. Pokud by se dalo nastavit implicitní vypnutí, mělo by to v počátečních fázích nasazení značné výhody.

5.4 Priorita požadavků

V této části je již jasné, že požadavků na software není málo a bylo při nejmenším vhodné dát jim určitou prioritu a analyzovat, které mají přednost před jinými.

Nejdříve je nutné implementovat část spouštějící pluginy a část, která je schopna udržovat a načítat nastavení. Následovat by měla implementace datové komunikace mezi pluginy a jádrem aplikace. Dále by se měl zhotovit mechanismus pro zobrazení a analýzu dat. Tedy i nějaké grafové výstupy.

Implementace samotných pluginů bude až poslední fází. Všechny pluginy mají prakticky stejnou hodnotu priority až na pluginy, které by měli zabezpečovat vhodné pojmenování tříd a proměnných. Přesto by bylo vhodné již teď několik vybrat, které budou implementovány nejdříve, aby mohlo dojít co nejdříve k integraci a testování na reálných souborech pracovního projektu. Proto budou mít jednoduše přednost ty, které se v seznam v předchozí kapitole vyskytují na předních pozicích.

6 Návrh

6.1 Návrh jádra

6.1.1 Obecný slovní popis

Aplikace bude primárně určena na operační systémy unixového typu (Linux, Solária a Mac OS X). Samotný program bude obsahovat obsluhu pouze přes příkazovou řádku. Jako programovací jazyk jádra byl vybrán C++. Ke svému běhu bude potřebovat i aplikaci PHC v nejnovější verzi a to tedy 0.1.7. Dále pak interpreta PHP ve verzi 5 nebo vyšší

Jako formát konfiguračních souborů bylo vybráno XML pro svou obecně velmi dobrou čitelnost a relativně pohodlnou modifikaci. Nevýhodou bude implementace parseru, která není triviální, ale je zvládnutelná v relativně rychlém čase. XML bude také formátem pro výměnu dat mezi pluginy. Konfigurační soubory bude mít aplikace hned dva. První bude obsahovat přímo nastavení aplikace, jako jsou adresy externích spustitelných souborů a jejich parametry pro aktivaci pluginů. V neposlední řadě by měla obsahovat adresu aplikace, která se postará o odeslání emailu s popisem vyhledaných chyb.

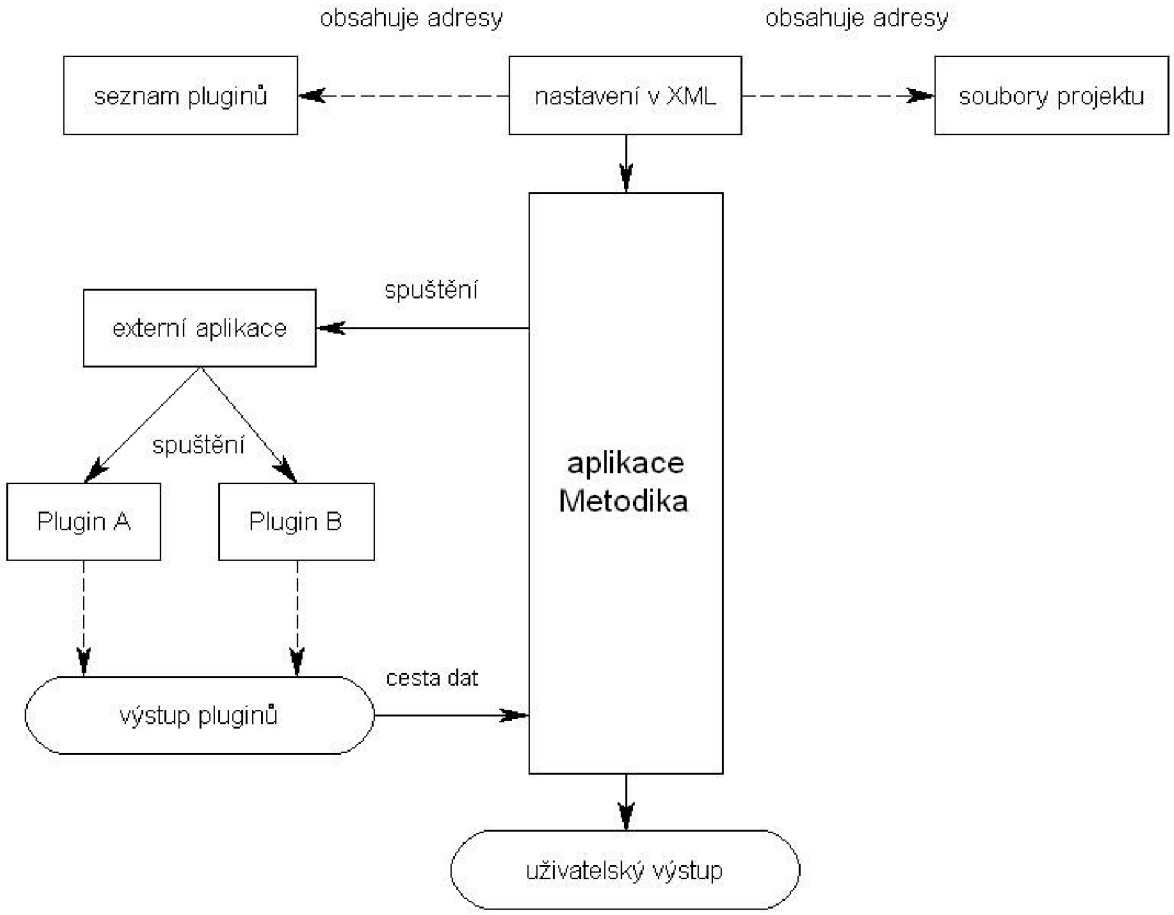
Další XML soubor bude velmi často editován a bude popisovat nastavení k samotnému projektu. Bude obsahovat informace o adresářové struktuře. Přesněji bude obsahovat hlavní adresář projektu. Poté si aplikace již sama načte všechny podporované soubory, které v tomto adresáři existují. Seznam přípon podporovaných souborů bude také součásti XML souboru. Součástí může být také definice souboru jednotlivě. V tomto souboru bude možné definovat i typ a formát výstupu. Následovat bude nastavení a seznam pluginů. Každému pluginu bude možnost poslat určitou sadu parametrů a zakázat mu přístupy do vybraných adresářů nebo souborů. Kromě těchto základních nastavení bude uživatel moci nastavit typ hodnoty výsledku pluginu. Vyhledaný vzor může být klasifikován jako neviditelný, jako informace, varování nebo hrubá chyba. Poslední informací, kterou tento soubor ponese bude seznam identit. V praxi tedy programátorů, kteří jsou zařazeni k projektu. Každý zde bude mít svoje jméno, jednoznačný identifikační řetězec a kontaktní emailovou adresu, na kterou mu můžou chodit informační emaily.

Z konfiguračních souborů je zřejmé, že aplikace bude soubory načítat automaticky a data předávat pluginům. Bude možné předat standardní vstup nějakému určitému pluginu nebo všem právě nastaveným v konfiguračním souboru.

Výstup bude klasicky textovou formou nebo formou HTML prezentace s grafovým zobrazením. O výstup se bude starat jádro systému, které před svým výstupem načte data, ze všech pluginů. Posledním možným výstupem bude zaslání emailu s informací, která bude podmnožinou textového nebo HTML výstupu.

Tento popis těchto souborů jistě není dostatečný. Přímo vstupu a výstupu se bude věnovat další podkapitola návrhu. Tam bude vše následně upřesněno a formálně definováno..

Po spuštění s vhodným parametrem (viz kapitola Vstup) se provede načtení XML souborů do paměťových struktur, které budou upřesněny diagramy tříd. Postupně se spustí na základě nastavení externí interprety a aplikace, které spustí pluginy Metodiky. Na pluginu již pak bude zabezpečit vhodný XML výstup, kterému bude jádro aplikace rozumět (opět kapitola Vstup). Kouzlo této myšlenky je v tom, že může být plugin prakticky jakéhokoliv typu. Rozumějme v kterémkoliv interpretovaném i klasickém jazyce. Důležité je, aby Metodika znala adresu souboru pro spuštění interpretu a plugin zabezpečil vhodný výstup. Následuje sbírání výstupu pluginů jako vstupů do Métodiky. Métodika vstupy vyhodnotí a podle parametrů s výsledkem i dál bude pracovat. Činnost jádra ilustruje následující obrázek č.13.



Obrázek 13 : Data v v aplikaci Metodika

V případě jiných voleb při spuštění se bude provádět určitá podmnožina činností z minulého odstavce nebo triviálnější úlohy, které budou ještě specifikované.

6.1.2 Vstupy

První je podstatné zmínit volby při spuštění z příkazové řádce. Pro implementaci výběru chování bude využita standardní funkce pro řízení parametrů „getopt“.

metodika **[VOLBA]** ... [SOUBOURY] ...

-h	Vytiskne nápovědu a ukončí se
-p	Jako parametr figuruje adresa ke konfiguračnímu XML projektu
-s	Načte samotný plugin a aplikuje ho no všechny soubory projektu
-i	Jako parametr figuruje unikátní řetězec identity a touto volbou programu uživatel řekne kdo je vlastníkem právě zpracovávaných úprav

V případě neprázdného standardního vstupu se vstup počítá jako jediný zdroj PHP kódu. Po spuštění se načte automaticky XML soubor s obecným nastavení. Jeho DTD (tedy anglicky Data Type Definition) je uvedeno na následujících řádcích :

```
<!ELEMENT metodica (types, emails)>
<!ELEMENT types (type)>
<!ELEMENT type (compiler, exec)>
<!ATTLIST type name CDATA >
<!ELEMENT compiler (app, (parameter)*, suffix)? >
<!ELEMENT exec (app, (parameter)*, suffix)>
<!ELEMENT emails (app, (parameter)*)>
<!ELEMENT type (compiler, exec)>
<!ELEMENT app CDATA>
<!ELEMENT parameter CDATA>
<!ELEMENT app CDATA>
<!ELEMENT suffix CDATA>
```

Pro příklad zde může být definován jako “type” interpret jazyka PHP a jako parametry potom způsob, jakým se volá na zdrojový soubor určitý plugin. Jako pomocné konstanty zde figurují [pluginpath], která je nahrazena relativní cestou k pluginu a [file], která je nahrazena relativní cestou k souboru se zdrojovým kódem, který je třeba analyzovat.

U podřízených značek značky „emails“ se vyskytují i jiné konstanty. Jsou to [address], která je nahrazená internetovou adresou přihlášené identity. [subject] je nahrazena předmětem zprávy a konečně [message], která figuruje jako samotný obsah emailu.

Dalším druhem XML souboru, který bude vstupovat do aplikace Metodika je soubor s nastavením samotného projektu. Opět pro detailní ilustraci možného obsahu a formátu je pro návrh na následujících řádcích DTD tohoto souboru :

```
<!ELEMENT project (files, plugins, output, identities)>
<!ATTLIST project name CDATA >
<!ELEMENT files (exclude, (suffix)+)>
<!ATTLIST files basedir CDATA >
<!ELEMENT exclude ((directory)*,(file)*)>
<!ELEMENT plugins (plugin)* >
<!ELEMENT plugin ((parameter)*,exclude?) >
 <!ATTLIST plugin type CDATA >
<!ATTLIST plugin file CDATA >
<!ATTLIST plugin auto (on | off) >
<!ELEMENT output (type, template)>
<!ELEMENT identities (identity)+ >
<!ELEMENT identity (id, name, email)>
<!ELEMENT suffix CDATA >
<!ELEMENT directory CDATA >
<!ELEMENT file CDATA >
<!ELEMENT parameter CDATA >
<!ATTLIST parameter name CDATA >
<!ELEMENT template CDATA >
<!ELEMENT type (HTML | text) >
<!ELEMENT id CDATA >
<!ELEMENT name CDATA >
<!ELEMENT email CDATA >
```

Nejvýznamnější atributem je zde jednoznačně atribut “basedir”, který obsahuje domovský adresář projektu. Řada značek s názvem “suffix” následně definuje přípony souborů, které mají v tomto adresáři a případných podadresářích akceptovat. Tag “exclude” pak obsahuje seznam souborů a adresářů, které se mají vyjmout, tedy přesněji nezpracovávat. Dále zde figuruje seznam pluginů s adresami (tag „file“) a jejími typy (viz předchozí kapitola). U každého může být také zapnuté exkluzivní vypnutí atributem auto nastaveným na hodnotu „off“. Každý plugin může mít seznam

souborů, pro které se nemá používat a neposledně také seznam parametrů. Ty následně slouží k předávání informací přímo do pluginu. Takto můžeme specifikovat povinnou předponu tříd u Pluton; který má za úkol tuto činnost a podobně. Typ výstupu již byl zmíněn několikrát a tedy není třeba se dále rozšiřovat. Poslední rodinou tagů je seznam identit. Ty byli už také zmíněny a mají hlavní účel přiřadit správnou emailovou adresu při odesílaní emailu.

Posledním typem XML souboru, který vstupuje do aplikace Metodika je výstup z pluginů. I zde pro ilustraci výčet DTD elementů:

```
<!ELEMENT output (file)*>
<!ATTLIST output project CDATA >
<!ELEMENT file (pattern)+)>
<!ATTLIST file name CDATA >
<!ELEMENT pattern (name, line, solution)>
<!ATTLIST pattern name CDATA >
<!ELEMENT line CDATA>
<!ELEMENT solution CDATA>
```

Zde je definice nejednoduší. Ve zkratce jde totiž jen o seznam souborů a pod každým souborem může existovat seznam všech vyhledaných metodických chyb. Vzor má atribut unikátní řetězec "name", dále elementy pro číslo řádku na kterém chybový vzor začíná. Následují dva delší řetězce s popisem daného vzoru a možnosti jeho řešení.

6.1.3 Výstupy

Výstupem jádra programu je vždy seznam pětic ve formátu, které si uživatel zvolil v konfiguračním XML souboru. Výstupem je tedy vždy unikátní název, číslo řádku, popis problému, jeho řešení a případně URL do online nástěnky.

Grafový výstup v rámci HTML výstupu bude zajišťovat PHP knihovna „JpGraph", která zvládne zobrazování široké palety grafů. Knihovna má velmi snadné aplikační rozhraní a výstupem je vždy obrázek, takže výstup není limitován pouze na webové stánky.

6.1.4 Adresářová struktura

Konfiguračních a vůbec i ostatních souborů, které vstupují a vystupují v rámci aplikace Metodika není málo, proto bude lépe jednoznačně, už v návrhu definovat rozložení souborů do adresářů. Definovat jejich názvy a ušetřit si tak v implementaci možné problémy.

metodika/data/	Adresář obsahující většinu XML dat

metodika/data/include/	Adresář obsahuje podporu pro pluginy
metodika/extern/	Adresář obsahuje podporu pro pluginy
metodika/plugins/	Adresář obsahující všechny pluginy
metodika/output/	Adresář pro XML výstupy pluginů
metodika/metodika	Spustitelný soubor, přístupový bod
metodika/data/settings.xml	Hlavní nastavení metodiky
metodika/data/project.xml	Projektové nastavení metodiky
metodika/data/include/metodika.php	Podpora pro pluginy typu PHP
metodika/output/out.xml	Soubor s posledním výstupem pluginů
metodika/data/plugins.input	Alternativní vstup pluginů, seznam parametrů
metodika/data/plugins.source	Alternativní vstup zdrojového PHP kódu

6.2 XML parser

O nutnosti implementace XML parseru již v této práci padla zmínka. Parser bude implementován jako klasický stavový automat a postupně bude kód procházet. Jednoprůchodově XML souborů se bude dostávat do různých stavů a to pak ovlivní jeho chování. Seznam konstant pro stavy spolu s krátkým popisem :

_START	Startovací tag, hledá počáteční značku
_TAG	Hledá standardní tag
_TAG_OPEN	Stav, kdy je parser uvnitř počátečního tagu
_TAG_CLOSE	Stav, kdy je parser uvnitř ukončovacího tagu
_VALUE	Parser je ve stavu mezi otevíracím a uzavíracím tagem
_ATTRIBUTE	Stav, kdy se právě čte název atributu
_ATTRIBUTE_VALUE	Stav, při kterém se čte hodnota současného atributu
_ERROR	Chybový stav v případě neočekávaného znaku

Podrobný návrh obslužné třídy bude v následující kapitole. Ještě před návrhem je však vhodné zamyslet se nad přístupem k jednotlivým hodnotám tagů a hodnotám atributů. Inspiroval jsem se přístupem, který využívají metody v PHP pro funkci s DOM modelem a rozhodl jsem se, že by bylo chytré využít asociativního kontejneru map v jazyce C++.

Přístup k hodnotě nějakého zanořeného tagu nebo atributu by mohl vypadat následovně :

```
xml->nodes[„tag1“]->nodes[„tag2“]->value
xml->nodes[„tag1“]->nodes[„tag2“]->attributes[„nazev“]
```

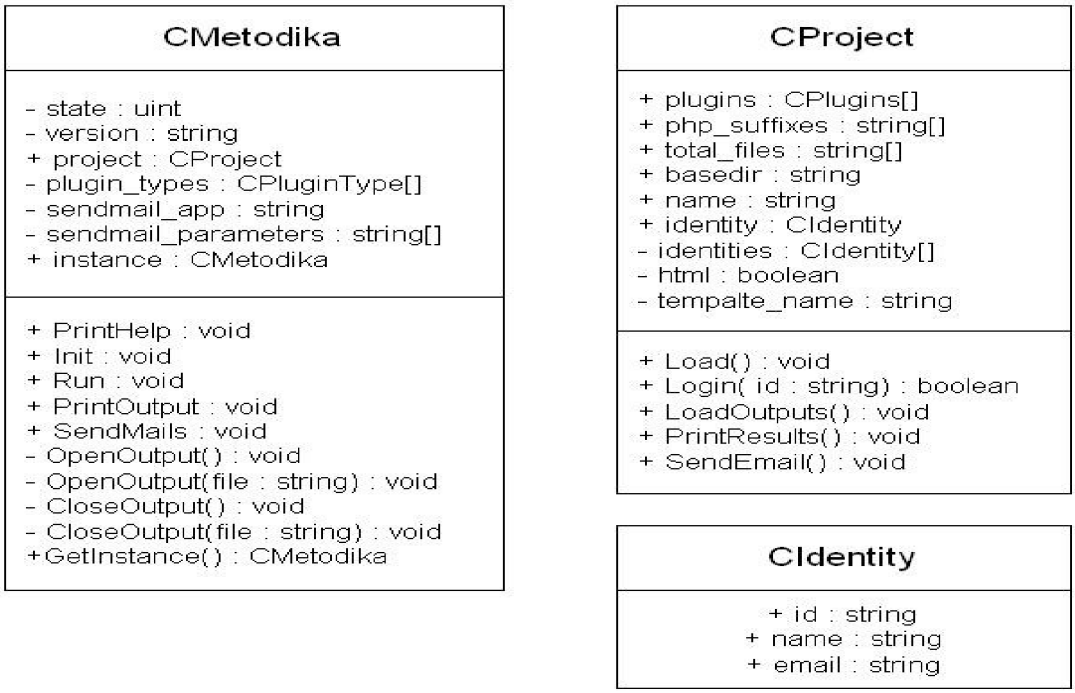
Zanoření by prakticky nebylo limitováno a obrovskou výhodou by byla přehledná práce, která by do značné míry kopírovala hierarchii XML souboru.

6.3 Diagramy a popis tříd

6.3.1 CMetodika, CProject a CIdentity

Třída CMetodika zaštiťuje základní nastavení aplikace a podporuje sadu elementárních operací nad programem. V tomto případě byl využit pro tuto třídu speciální návrhový vzor a to konkrétně singleton. Singleton je návrhový vzor, který zajišťuje pohodlný a unikátní přístup k instanci hlavní třídy. Metody Run, PrintOutput a SendEmails volají stejně pojmenované metody u projektů nebo volají tyto metody u všech pluginů přiřazeného projektu. Třídy OpenOutput a CloseOutput jsou přetížené. Ta bez parametru volá otevření výstupního XML souboru s počátečným tagem. S parametrem je pak volána pro každý plugin.

CProject je třída, která zaštiťuje projekt jako celek. Metoda Load načte XML soubor a naplní jim podobně pojmenované datové položky. LoadOutputs spolu s PrintResults se postarají o manipulaci s daty od pluginů. PrintResults tiskne výsledek projektu. Metoda Login se stará o přiřazení správné pracovní identity, což znamená že prohledá pole se všemi identitami a pokud se shoduje identifikační řetězec, naplní vlastnost objektu identity správnou identitou programátora.

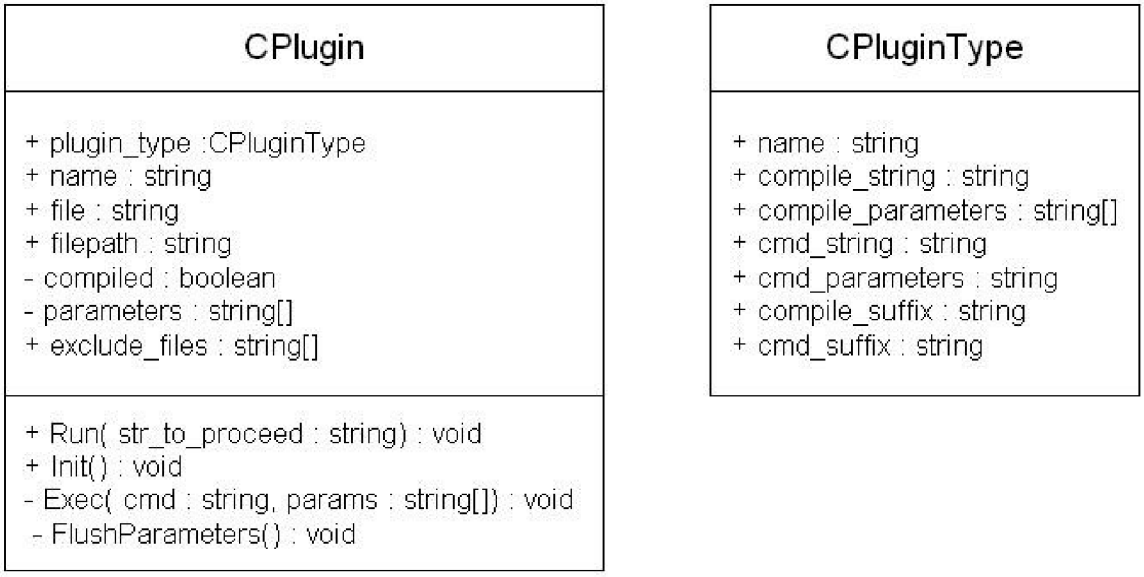


Obrázek 14 : Digramy tříd CMetodika, CProject, CIdentity

6.3.2 CPlugin, CPluginType

Další dvě třídy jsou abstrakcí nad pojmem plugin. CPlugin zaštiťuje plugin jako spustitelnou část kódu, která má své vlastnosti a operace. CPluginType říká, jak se k danému pluginu má chovat externí aplikace (myšleno ve většině případů interpret).

Metoda Run bude zajišťovat volání externího programu na plugin a případnou jeho kompilaci pokud tak plugin potřebuje. V neposlední řadě se bude starat o nahrazení konstant v hranatých závorkách, které vstupují do aplikace skrze základní XML s nastavením Jedna z metod, která je volána, je metoda Exec. Ta bude mít na starosti vytvoření nového procesu a spuštění externí aplikace zadané pomocí prvního parametru. Metoda FlushParametrs naplní soubor alternativního vstupu pro pluginy textovým seznamem parametrů. Jako poslední zbývá metoda Init, která na základě přípony pluginového souboru určí, co se jedná za typ pluginu a naplní vhodné atributy objektu základními hodnotami.

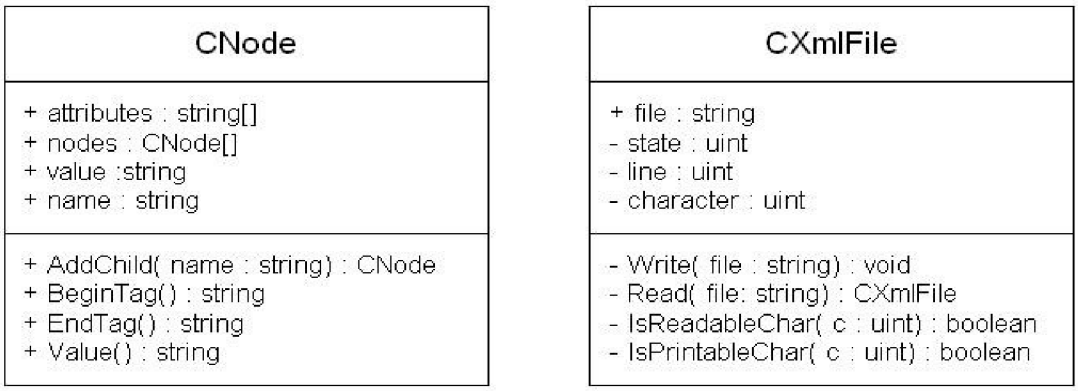


Obrázek 15 : Digramy tříd CPlugin, CPluginType

6.3.3 CNode, CXmlFile

Následují dvě třídy pro zpracování XML souboru. Hlavní třídou je zde CXmlFile, která dědí z třídy reprezentující uzel CNode. Navíc však osahuje dvě pomocné metody a metody pro podporu zápisu a čtení dat souboru. Tato třída má v oblasti uzlů vedoucí úlohu a je brána jako kořenový uzel XML dokumentu.

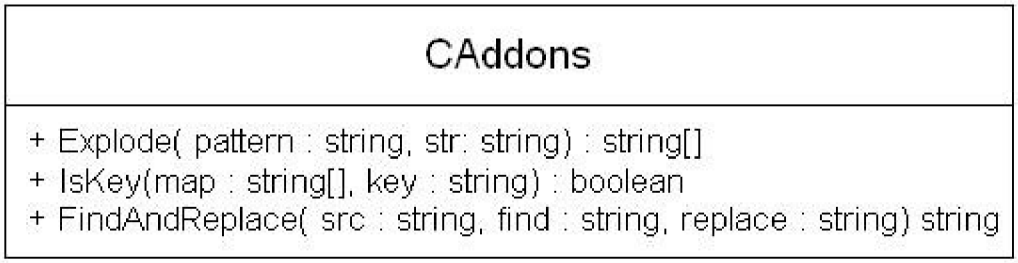
CNode reprezentuje klasický uzel XML dokumentu. Vlastní tedy kolekci všech podřazených uzlů, vlastní hodnotu a seznam všech svých atributů spolu s hodnotami.Jak již bylo lehce naznačeno v minulé kapitole využije se výborných vlastností asociativního kontejneru „map“.



Obrázek 16 : Digramy tříd CNode, CXmlFile

6.3.4 CAddons

Tato třída slouží k zapouzdření obecných metod pro práci s řetězci a kontejnery.. Metoda Explode pracuje tak, že rozdělí podle určitého vzoru řetězec do kontejneru. Tímto způsobem lze například rozdělit věty v článku pomocí teček do pole. Další metoda FindAndReplace již podle jména dělá náhradu jistého podřetězce. Poslední metoda IsKey pracuje s kontejnerem map a dokáže zjistit zda v něm existuje daný klíč.



Obrázek 16 : Digramy třídy CAddons

6.4 Pluginy

6.4.1 Vstupy

Většina vstupů je pluginu předávána skrze parametry jejich externích interpretů či aplikací, které se starají o jejich bezproblémové spouštění. Další možnost již byl zmíněna a to skrze alternativní soubor s parametry formát je velice jednoduchý a o načítaní se stará sám plugin :

nazev_parametru=hodnota parametru

Součástí vstupu skrze parametry interpretů je adresa souborů jenž se má analyzovat pluginem. Pokud je prázdná, plugin musí hledat zdrojový kód v soboru popsaném jako alternativní vstup PHP kódu v kapitole 6.1.4

6.4.2 Výstupy

Výstup pluginu si vždy zabezpečuje sám plugin. Je třeba, aby dodal skupinu značek počínaje značkou „pattern“ do souboru pro výstupy pluginů. Vše nutné k návrhu ohledně výstupu již bylo zmíněno v předchozích kapitolách.

6.4.3 Pluginy pro PHC

Pro začátek vývoje se počítá se dvěma hlavními typy pluginů PHP a PHC. PHC jako každý jiný plugin musí splňovat požadavky na vstup a výstup uvedené v předchozích kapitolách. Pro tento typ pluginů se v první fázi vývoje Metodiky, což znamená v rámci této práce, nepočítá žádná podpora. Ve výsledku to znamená jen to, že si všechny komunikace s daty musí zabezpečit každý plugin zvlášť. Unikátní pro tento typ pluginů je, že můžou využívat API frameworku PHC. Konkrétně jde o třídu TreeVisitor. Tomuto frameworku byla věnována celá jedna kapitola a dělá tento typ pluginu velice mocným nástrojem.

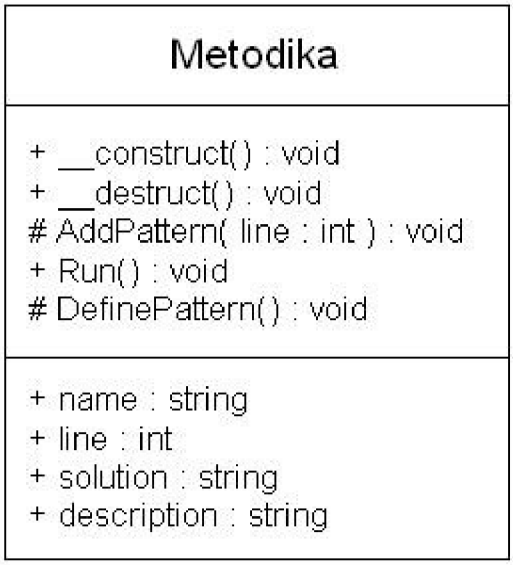
6.4.4 Pluginy pro PHP

Pluginy pro PHP zaostávají svou silou v možnosti jednoduché analýzy syntaktického stromu zdrojového kódu. Na žádost kontaktní osoby pracovní skupiny se bude jednat o hlavní typ pluginů implementovaných v rámci této práce.

Aby tvorba tohoto druhu pluginů byla jednoduší, bude vytvořena třída, z které budou následně všechny pluginy typu PHP dědit. Třída zabezpečí vstup-výstupní operace a naznačí jak by měla vypadat objektová obálka okolo tohoto typu pluginů. Více v další kapitole.

6.4.5 Podpora pro pluginy

Jak bylo již řečeno v minulé kapitole. podpora pro pluginy je ve formě otcovské třídy. Na obrázku č.17 je pro lepší pochopení diagram této třídy.



Obrázek 17 : Digramy třídy CAddons

Třída obsahuje všechny důležité atributy, definovaný konstruktor, který se stará o inicializaci proměnných a v neposlední řadě načtení volitelných vstupních parametrů do pole. Kromě konstruktoru je definovaný i destruktor, který se sám postará o v hodné uložení vyhledaného chybového vzoru. Stačí pouze zavolat metodu AddPattern, která jako první a zároveň jediný parametr bere číslo řádku a vše je hotovo. Metoda DeffinePatter a Run jsou deklarovány jako abstraktní a předcházejí tak možné implementační chybě. DefinePattern definuje chybový vzor a Run se stará automaticky o provádění akce pluginu.

V následující kapitole bude seznam navržených pluginů. Názvy budou v anglickém jazyce, jako kdyby se jednalo a třídy. Výstup jednotlivých pluginů bude samozřejmě v českém jazyku. Bylo by matoucí pojmenovávat třídy pluginu v českém jazyce, když třídy zaštiťující jádro aplikace jsou také v angličtině.

V řadě PHP pluginů se využije mnoho vestavěných PHP funkcí, které dokážou ušetřit mnoho práce. Jednou z nejmocnějších funkcí je „token_get_all“, která jako první a jediný parametr přijímá řetězec, který se má zpracovat. Funkce následně provede lexikální analýzu a rozdělí vstupní řetězec obsahující zdrojový PHP kód na lexémy. Funkce tedy vrací vícerozměrné pole tokenů obsahující v prvním indexu konstantu určující typ. K této funkci jsou deklarovány konstanty reprezentující typ tokenu a mají prefix „T_“. Seznam je poněkud obsáhlý a není jej třeba zde zveřejňovat. Je ovšem dostupný na přiloženém optickém mediu. Na druhou stranu již konkrétní typy jsou důležité pro každý plugin zvlášť, proto každý plugin ve svém popisu bude obsahovat i seznam typů tokenů, se kterými

bude pracovat. Kromě typu je zde hodnota tokenu, což bývá v naprosté většině přímo hodnota řetězce, který je přiřazena tokenu. Posledním prvek v poli je číslo řádku, na kterém se token v kódu nacházel.

6.4.6 Plugin PhpVoid

Tento PHP plugin bude implementován jen tak, že bude dědit z třídy Metodika. Nebude mít žádnou funkci a bude jen demonstrovat implementaci. Pomůže tak testování a případnému následnému vývoji v pracovní skupině.

6.4.7 Plugin PhcVoid

Tento plugin je na rozdíl od předchozího implementován v jazyce PHC. Stejně jako PhpVoid nebude mít žádnou funkci a bude jen demonstrovat implementaci a pomůže budoucí implementaci pluginů pod touto platformou.

6.4.8 Plugin NoTab

Tento PHP plugin bude provádět požadavek definovaný v kapitolo 5.3.1. Nejdříve bude zpracován kód na tokeny a v tokenu pod konstantou T_WHITESPACE (obsahuje neviditelné znaky) se bude hledat výskyt znaku tabulátoru. V případě výskytu se volá metoda AddPattern.

6.4.9 Plugin ForbidOutput

PHP plugin se vztahuje k požadavku 5.3.2. Ve zpracovaném kódu bude vyhledávat všechny tokeny označené jako T_ECHO a T_PRINT. V případě výskytu bude volána metoda k vypsání chybového vzoru. Je nutné kontrolovat i další funkce, které generují výstup. Jsou jimi funkce var_export, var_dump a print_r. Ty bude plugin hledat v tokenech označených jako T_STRING. V tomto tokenu se nacházejí mimo jiné všechny volání externích funkcí.

6.4.10 Plugin ForbidDebugOutput

Tento plugin bude pracovat podobně jako předcházející, jen bude vyhledávat funkce sloužící k výstupu pouze v tokenu T_STRING

6.4.11 Plugin OnlyEcho

Plugin OnlyEcho prochází pole tokenů a pokud najde typ T_PRINT ihned zahlásí chybový vzor. Dále bude hledat token T_ECHO a kontrolovat zda se za ním nachází token T_WHITESPACE s délkou v

něm obsaženém řetězci rovnu jedna. Pokud tomu tak skutečně bude, bude se tento plugin krýt s požadavkem 5.3.4.

6.4.12 Plugin PhpTags

Tento plugin bude hledat v poli tokenů typy T_OPEN_TAG , T_OPEN_TAG_WITH_ECHO a T_CLOSE_TAG. Tyto tokeny obsahují uzavírací a otevírací PHP řetězce. Plugin přesně ve shodě s požadavkem 5.3.5 zkontroluje zda se hodnoty těchto tokenů rovnají zadání. V opačném případě se volá metoda AddPattern.

6.4.13 Plugin MaxLength

PHP plugin MaxLength bude velice jednoduchý. Jako parametr do něj bude vstupovat maximální počet znaků na řádku a plugin tento fakt zkontroluje se skutečnou délkou řádky.

6.4.14 Plugin ForceReturn

Plugin bude pracovat ve shodě s požadavkem z kapitoly 5.3.7. Postupně bude procházet pole tokenů a jakmile najde token T_FUNCTION přejde do stavu, kdy bude čekat token s hodnotou otevřené složené závorky. Plugin bude obsahovat čítač složených závorek a v případě, že klesne na nulu, tedy počáteční hodnotu, znamená to jediné a to konec funkce Pokud během provádění čítaní nebude nalezen token T_RETURN bude zahlášena chyba.

6.4.15 Plugin CurlyOnLine

Tento plugin prochází pole tokenů pro hodnoty otevřené nebo uzavřené závorky. Jakmile najde takovou shodu, následuje kontrola zda nepředcházel token T_WHITESPACE, který obsahoval znak nového řádku. Jestli tomu takto není bude hlásit chybu. Takto dosáhneme funkčnosti podle požadavku 5.3.8.

6.4.16 Plugin CaseFall

Abychom dosáhli funkčnosti shodné s požadavkem 5.3.9 je třeba vyhledat konstrukci switch, která je v polí tokenů typem T_SWITCH a následně kontrolovat sekvence T_BREAK a T_CASE. Pokud najdeme stejnou dvojici, je nutno hledat jestli předchozíjí token je typu komentáře, tedy T_ML_COMMENT nebo T_COMMENT.

6.4.17 Plugin ArrayIndex

Tento plugin bude pracovat na základě požadavku 5.3.10. Prohledá pole tokenů pro hodnoty hranatých závorek, které symbolizují indexaci polí. Následovat nebo předcházet takovému tokenu

nesmí v žádném případě token typu T_WHITESPACE. Pokud je jako index pole token T_CONSTANT_ENCAPSED_STRING, zkontroluje se zda jeho obsah začíná a zároveň končí jednoduchými uvozovkami. V neposlední řadě se obsah tohoto tokenu podrobí analýze zda-li neobsahuje nepovolené znaky, které mohou vyvolat interní varování

6.4.18 Plugin ArrayComas

Plugin je navržen tak, aby pracoval ve shodě s požadavkem v kapitole 5.311. Postupně prohledává pole tokenů pro token typu T_ARRAY. V případě nálezu si hlídá počet otevřených a uzavřených kulatých závorek a pokud na konci nenásleduje token s hodnotou čárky bude nahlášena chyba pomocí metody AddPattern.

6.4.19 Plugin SpacedIf

PHP plugin SpacedIf bude hledat tokeny učené konstantami T_IF, T_SWITCH, T_WHILE, T_FOR a T_FOREACH. V případě nálezu libovolného z těchto tokenů se plugin podívá o jeden token v před a pokud se bude jednat o token typu T_WHITESPACE s hodnotou obsahující pouze jediný prázdný znak, je vše v pořádku. V opačném případě tento plugin musí zahlásit nález chybového vzoru.

6.4.20 Plugin SpacedOperators

Tento plugin bude pracovat velice podobně jako předcházející. Navíc však bude hledat onu „mezeru“ i před tokenem. Seznam tokenů, s kterými bude pracovat je navíc úplně jiný a poněkud rozsáhlý. Jedná se o tokeny popisující všechny operátory : T_BOOLEAN_AND, T_BOOLEAN_OR, T_IS_GREATER_OR_EQUAL, T_IS_IDENTICAL, T_IS_IDENTICAL, T_IS_NOT_EQUAL, T_IS_NOT_IDENTICAL, T_IS_SMALLER_OR_EQUAL, T_IS_EQUAL, T_LOGICAL_AND, T_LOGICAL_OR, T_LOGICAL_XOR, T_MINUS_EQUAL, T_MOD_EQUAL, T_MUL_EQUAL, T_OR_EQUAL, T_PLUS_EQUAL, T_SL,T_SL_EQUAL, T_SR, T_SR_EQUAL, T_XOR_EQUAL, T_AND_EQUAL. Kromě těchto typů se ještě budou kontrolovat hodnoty pro řetězce "+", "*", "/", "-", "%", "=".

6.4.21 Plugin CommentedCurly

Tento plugin se bude snažit splnit požadavek 6.3.14 hledáním tokenů s hodnotami uzavřených složených závorek. V případě nálezů tří takových tokenů neoddělených tokeny T_ML_COMMENT nebo T_COMMENT se bude volat metoda AddPattern.

6.4.22 Plugin QuestionNotation

Tento plugin souvisí s požadavkem 6.3.15. Bude prohledávat pole tokenů pro dvojici typů T_IF, T_ELSE. Jakmile takovou posloupnost najde, bude hledat mezi tokeny s hodnotami složených

závorek středníky. Pokud najde pouze dva, upozorní výsledného programátora tento plugin metodou AddPattern.

6.4.23 Plugin ForceCurly

Plugin ForceCurly hlídá podle jednoho z požadavků zda každá konstrukce if má vlastní složené závorky. Podobně jako předchozí plugin hledá výskyt tokenu T_IF a T_ELSE a hledá za nimi okamžitý výskyt otevřené složené závorky. Ze syntaxe jazyka je jasné, že složená závorka následuje až za párem kulatých závorek.

6.4.24 Plugin ClassPrefix

PHP plugin ClassPrefix bude velice triviální a jen zkontroluje zda v hodnotě všech tokenu T_CLASS existuje jako první znak „C“ nebo v případě zadaných alternativních parametrů porovná předponu s parametrech s názvem „prefix“.

6.4.25 Plugin IncludeCheck

Požadavek v kapitole 5.3.19 má za následek návrh tohoto pluginu. IncludeCheck prohledá pole tokenů pro konstanty T_INCLUDE, T_INCLUDE_ONCE, T_REQUIRE_ONCE a T_REQUIRE. Za nimi, po případném tokenu T_WHITESPACE, se nesmí vyskytovat ani jedna ze superglobálních proměnných kromě proměnné globals. Jak již bylo zmíněno v požadavku, jedná se i o bezpečnostní riziko.

6.4.26 Plugin GlobalList

V souladu s požadavkem 5.3.20 bude tento plugin přijímat seznam nepojmenovaných alternativních parametrů, které mu poskytnou povolený seznam globálních proměnných. Seznam skutečně použitých globálních proměnných si plugin zjistí průchodem polem tokenů, kde bude hledat následující proměnnou za tokenem označeným typem T_GLOBAL .Výhodou v PHP je, že před použitím globálních proměnných v souboru nebo dokonce metodách a funkcích musí dojít k opětné deklaraci všech globálních proměnných, které se programátor chystá použít.

6.4.27 Plugin ElseIfOnLine

Tento plugin bude jednoduše vyhledávat v poli tokenu dvojici T_ELSE a T_IF v tomto pořadí. Tokeny musejí být navíc odděleny tokenem T_WHITESPACE s hodnotou řetězce, která bude délky jedna a obsahovat pouze prázdný znak.

6.4.28 Vypinání a zapínání pluginů

O detekci vypínání a zapínání pluginů se budou pluginy starat samy. Implicitně budou všechny zapnuté, pokud jim jádro nesdělí pomocí parametrů jinak. Samotné zapnutí a vypnutí se bude hledat opět v poli tokenů, kde se v případě nálezu tokenu T_COMMENT bude ověřovat hodnota s následujícími dvěma variantami :

//METODICA:on
//METODICA:off

Pokud budou tyto komentáře nalezeny, musí plugin pomocí vlastní implementace přejít do stavu, kdy bude zpracovávat či nezpracovávat zdrojový kód.

6.5 Nástěnka

Publikace metodických pravidel pro programátorský tým proběhne skrze vlastní internetovou stránku. Stránka bude implementována na zázemí, které poskytuje technologie MediaWiki. Adresa a i další náležitosti budou vybrány konzultantem této práce Ing. Michal Juroszem, který má v této oblasti dostatečná práva.

Obsah nástěnky bude doplněn seznamem všech pluginů, které se budou aplikovat na stávající i nový kód. U každého pluginu bude vždy již známá identifikace pluginu s jeho popisem a dalšími parametry. Kromě těchto atributů bude také obsahovat ukázku zdrojového kódu, který splňuje kriterium implementované daným pluginem a zároveň kód, který ho porušuje.

7 Implementace

7.1 Úvodem

Implementace proběhla plynule a relativně velmi rychle v řádu několika týdnů. Velmi pomohla detailní příprava a návrh. Díky tomu bylo využito již implementovaných metod v jazyce PHP . Mezi nimi byly funkce pro lexikální analýzu a pohodlnou manipulaci se vstup-výstupními operacemi.

Priorita požadavků následně pomohla upřednostnit implementaci některých částí dřív a tak mohlo dojít k testování ještě neúplné aplikace na reálných datech, tedy přímo souborech informačního systému VUT, které prokázali celkovou funkčnost aplikace.

7.2 Implementační nástroje

Implementace byla prováděna na operačním systému Ubuntu v.7. Ke kompilování C++ kódu byl použit překladač g++. Jako vývojové prostředí jak pro PHP a C++ bylo využitá platforma Eclipse. Pro grafický návrh diagramu tříd bylo využito jednoduchého nástroje RF-Flow.

7.3 Změny oproti návrhu

Příliš mnoho změn oproti návrhu se při implementaci nedostavilo. Změny, ale můžeme najít v původních ´úvahách pro realizaci pluginů. PHP funkce get_token_all se nakonec ukázala natolik mocná, že byla použita v naprosté většině pluginů. Důvodem je, že řešené problémy nejsou algoritmicky příliš složité a nebylo na ně třeba brát mocnější nástroj, který poskytuje platforma PHC. Jádro aplikace Metodika přesto tuto platformu podporuje a tato práce obsahuje mnoho teoretických informací, které jistě pomohou návrhu a implementaci dalších pluginů implementovaných pod touto platformou. Není tedy třeba chápat přílišné nevyužití frameworku PHC jako nevýhodu, ale jako otevřenost pro jednoduché rozšíření a změny v případě nových požadavků

8 Testování, distribuce a integrace

8.1 Samostatné testování

8.1.1 Specifikace testů

Před dokončení několika finálních úprav bylo provedeno samostatné testování na fiktivních a reálných datech. Tyto testy budou následovně předány kontaktní osobě vývojového týmu s nimž je tato práce úzce svázána. V prvé řadě mají otestovat funkčnost v praxi a podat informace o současném stavu zdrojových kódu aplikací přiřazeného vývojového týmu.

Pro jednotkové testy byl vyvinut krátký skript, který všechny jednotky (pluginy) podrobí testům na fiktivních zdrojových kódech, které mají vždy za účel prověřit kvalitu pluginů. Zdrojové soubory byly označeny jako positivní v případě, že se má výsledek vyhodnotit kladně nebo naopak negativní. Skript v poslední řadě poskytne výstup ve formátu TAP (Test Anything Protocol). Následně bude testován náhodný vzorek ze systému jako integrační testování za účelem odstranění možných chyb, které běžné testování neodhalilo. Poté bude vzorek testován lidskou silou a oba výsledky porovnány. V případě shody bude následovat série reálných testů.

Dalším testem se zpracují veškeré soubory s příponou „php“ všemi pluginy, které byly implementovány. Jako chybové vzory byly nastavené pluginy pro kontrolu vkládaných souborů a plugin pro kontrolu počátečního PHP tagu. Ostatní vzory byly nastavené na varování. Výsledky bude nutné zhodnotit a rozdělit výsledky podle jednotlivých adresářů, které reprezentují různé části systému. Výsledek následně rozhodne, jakým způsobem budou nalezené chyby odstraněny. Soubor s výsledky je umístěn na přiloženém optickém mediu v adresáři „results“ v hlavním adresáři.

8.1.2 Testování jednotek a náhodné integrační testování

Výsledky jednotkových testů jsou spolu s testovacími soubory a skriptem umístěny na přiloženém optickém médiu.

Náhodný vzorek integračního testování ukázal, že pluginy skutečně pracují zcela shodně s návrhem. Jako náhodný vzorek byl zvolen soubor index.php v aplikaci portal3. Soubor měl dostatečně velký počet řádků a vykytovala se v něm široká paleta metodických chyb, které byly odhaleny. .

8.1.3 Výsledek pro jednotlivé adresáře

Adresář : _base

Popis : Adresář obsahuje základní datové soubory pro ostatní webové aplikace. Jedna se tedy o soubor knihoven, které jsou pak v aplikacích volně vkládány.

Počet souborů : 520

Počet varování : 13637

Počet chyb : 2929

V adresáři se vyskytuje široká paleta metodických chyb. V jednom souborů se používá hromadně tabulátor k odsazování v dalším už nikoliv. Někde se udržuje pravidlo, kdy jsou složené závorky na samostatném řádku. V tom samém souboru se v polovině pravidlo mění a je již dodržováno podle návrhu. Takhle je to prakticky se všemi chybovými vzory. Vyskytují se tu všechny bez jakékoliv pravidelnosti. Důvodem je, že na jednom souboru klidně pracovalo několik programátorů. Mezi několik málo vzorů, které se ve výsledku vyskytly v málem množství patří mezery mezi operátory a ukončení funkce „returnem“. Několik málo výskytů těchto vzorů přeci jen je, ale ve výsledku je třeba říci, že tyto pravidla jsou mezi programátory obecně rozšířená.

Dále je třeba se zamyslet nad vysokým čísle u počtu vzorů označených jako chyby. Jde o nesprávně označení kódu PHP tagem. Jak již bylo zmíněno je možné tuto chybu brát i mezi chyby v bezpečnosti a je třeba zjednat v krátkém čase nápravu.

Adresář : portal3

Popis : Adresář obsahuje data pro volně dostupnou prezentaci stránek informačního systému VUT Brno.

Počet souborů : 346

Počet varování : 72983

Počet chyb : 186

Výsledek se velmi podobá předchozímu. V podrobnějším rozboru souborů Metodiky lze najít prakticky stejný sled chyb. Opět se tu vyskytují ve větším množství porušení metodických pravidel složené závorky nebo nesprávného indexování pole.

Adresář : studis

Popis : Adresář obsahuje webovou aplikaci, která je určená pro studenty po přihlášení do informačního systému VUT.

Počet souborů : 180

Počet varování : 7822
Počet chyb : 46

Zde je výsledek pravděpodobně nejlepší ze všech tří testovaných aplikací. Pokud by jsme nepočítali adresář „template“ nevyskytuje se tady žádný chybový vzor otevřeného PHP tagu a tedy vzor, který je klasifikován jako chyba. Rozložení vzorů je také mírně odlišné od předchozích dvou aplikací. Vyskytuje se zde prakticky jen chybové vzory složených závorek na samostatném řádku a vzor, který je odhalován plugin pro kontrolu indexace polí. Ostatní chybové vzory se zde vyskytují opravdu minimálně.

8.1.4 Zhodnocení

Z testů v předchozí kapitole je jasně vidět, že systém trpí nedodržováním metodických pravidel. Prakticky, všechny implementované pluginy našli svůj chybový vzor. Testy ukázala, že „nejčistější“ aplikací je aplikace v adresáři studis. Mezi neobvyklejší nalezené chybové vzory byly nedodržování pravidla psaní složených závorek a různé typy indexování polí.

8.2 Integrace se SVN

Integrace proběhla jak bylo již řečeno v návrhu a s velkého procenta se o ní postaral Ing. Michal Jurosz, který má k nastavení Subverison serveru dostatečné práva a v neposlední řadě také zkušenosti.

Od zavedení tedy probíhá při tzv. „commitu“ verze souboru, tedy jinak řečeno potvrzení uploadu modifikace verze souboru na server, jedna činnost navíc. Subversion nebo programátor spustí aplikaci Metodika s parametrem -p a načte správně nastavený projektový XML s nastavením. Parametrem -i se zaregistruje správná identita, která právě commit vzdáleně spouští a jako standardní vstup se do aplikace dostane samotný zdrojový kód. Výstup bude skrze emailové rozhraní Metodiky poslán na email identity, která právě dělá funkci commit. Programátorovi tedy bude záhy doručen email s rozborem jeho zdrojového kódu. V případě, že se v jeho kódu vyskytnou chybové vzory je na ně upozorněn a měl by je co nejdříve odstranit. V případě, že takto neučiní lze viníka pohodlně skrze nástroje v Subversion manuálně dohledat.

8.3 Seznámení pracovní skupiny s nástrojem

Všichni členové vývojového týmu byly slovně seznámeni s nutností dodržovat jisté metodiky v rámci psaní zdrojového kódu. Každému členu pak byla slovně i emailem sdělena adresa s nástěnkou obsahující seznam nutných metodik. Programátoři byly v neposlední řadě seznámeni, že po odeslání

jejich kódu a commitu v rámci SVN dostanou vyrozumění ohledně metodické validity jejich zdrojového kódu.

Každý programátor se pak sám musí starat, aby metodiky dodržoval a tým z toho měl užitek. Metodika v tomto případě figuruje jako jakási policie, která hlídá zda se metodická pravidla skutečně dodržují.

9 Závěrem

9.1 Zhodnocení přínosu nástroje

V poslední kapitole je třeba lehce analyzovat přínos a význam této práce v rámci vývojového týmu VUT Brno. Teoretická práce zaměřující se na oblast statické analýzy PHP kódu zabrala více než 50% procent celkové času této práce. Takto rozsáhlá příprava byla nutná, aby bylo možné vhodně navrhnout a implementovat nástroj pro přiřazený vývojářský tým. Je jasné, že na finální analýzu bude potřeba více času a není zatím možné plně zhodnotit přínos na základě reálných faktů, ale je nutné v rámci této práce se o to pokusit.

Jednotnost způsobu zápisu dosavadního kódu se ukázala být jako nedostatečná a tato teoretická práce spolu s aplikací je prostředkem, jak tyto neduhy kódu odstranit a vnést do programování pořádek a jednotný styl.

Přestože byla v rámci této práce zpracována i online nástěnka, přesněji tedy webová stránka s metodickými pravidly, tak z již získaných dat lze říci, že existence nástroje pro kontrolu metodik na stránce je více než opodstatněná. Programátoři jsou jenom lidé a jako lidé i chybují. Přestože znají metodická pravidla, která je v rámci týmového úspěchu nutné dodržovat, není přechod bezbolestivý a často na nějaká pravidla zapomenou. Aplikace Metodika pracující jako policie v oblasti metodických zákonů uděluje včasné varování těm, kteří se prohřeší a předchází tak případnému opětovnému chaosu v kódu.

V rámci integrace byl programátorům rozdán dotazník, který má odpovědět na otázku zda nejsou některá pravidla příliš přísná a jak se jim zdá tato práce přínosná. Dotazník je k dispozici v rámci této práce jako příloha č.1. Slovní zhodnocení výsledků se nachází v následující kapitole.

9.2 Výsledky programátorského dotazníku

Z výsledku dotazníku vyplynulo, že ohlasy na potřebu zavést pravidla je vcelku kladná. Programátoři tedy ví, že jednotný kód má pomocí k lepší efektivitě programování.

Další otázkou bylo zda se jim zdá vidí potřebu udržovat metodiku externí aplikací. Více než dvě třetiny programátorů odpovědělo také kladně. Důvod této práce podle programátorů tedy existuje.

Poslední otázky měli zhodnotit názory na jednotlivá metodická pravidla a tedy zároveň na pluginy metodiky. Programátorům se vesměs zdály pravidla přínosná a vhodná. Nejméně oblíbeným pravidlem se stalo pravidlo psaní složených závorek na samostatném řádku. Tímto bylo prakticky zjištěno z čeho pramení spousta nálezných chyb v testování. Programátoři se jednoduše dělí na dvě skupiny. Jedna má raději složené závorky na samostatném řádku a druhá vždy jen pokud se jedná o koncovou. Je tedy nutné programátory v prospěch týmu přeučit k jiným programovacím návykům.

9.3 Pokračování vývoje

Další vývoj aplikace a obecně této práce byl diskutován s Ing. Michalem Juroszem. Nabízí se možnost překročení institutu VUT a tedy i vývojového týmu, pro který byl původně určen. Samozřejmě by se jednalo o volně dostupný produkt a bylo by vhodné zajistit vlastní samostatnou doménu. Případně i mírnou reklamní kampaň v podobně zveřejnění statistik a výsledků této práce. Dalším nápadem je i kompletně nový nástroj, který bude využívat jen PHP pluginy a jádro bude velmi odlehčené a bude též v jazyce PHP. Je známým faktem a dobrou myšlenkou programovat nástroj v jazyce, pro který je ve výsledku určen. Konkrétně by se jednalo o soubor skriptů s pluginy pro menší a střední vývojářské týmy. Ty by stačilo nahrát do projektové složky, spustit a zatrhnout volby metodik, na kterých by se tým shodl. Program by pak vygeneroval nástěnku metodiky v podobě HTML souboru a aplikoval by pravidla na vybrané adresáře projektu. Nejednalo by se tedy o automatickou činnost, ale o jakousi prověrku kódu v jednotlivých fázích. Jistá automatická kontrola by samozřejmě byla možná, ale byla by čistě v rukách vývojového týmu.

Otvírá se také možnost pouze nekontrolovat, ale chyby následně i automaticky opravovat. Nástroj by tedy sloužil i částečně k refaktorizaci. Nejsem si ale jistý, jaký by o takový nástroj byl zájem. Implementace by také nebyla triviální.

Rozšíření, které tento nástroj čeká téměř určitě je rozšíření počtu pluginu a tedy množství implementovaných metodických pravidel. A také úprava integrace v rámci systému Subversion, kde bude kontrola prováděná před samotným „commitem“ a zamezí se tedy publikování výsledného kódu s metodickými chybami.

Poslední možná cesta, kterou by se projekt mohl ubírat je implementace v rámci nějakého rozšíření PHP editoru. Nabízí se zejména nastroj Eclipse, který poskytuje velice dobrou podporu pro nástroje třetích stran nebo na druhé straně nástroj PHPED od firmy Nusphere. Ten je patrně nejpoužívanějším komerčním nástrojem v oblasti programování ve skriptovacím jazyce PHP.

Literatura

[1] Paleta Petr Co programátory ve škole neučí. Praha, Computer Press 2004 ISBN 80-251-0073-1

[2] Tansley, D. PHP a MySQL Vytváříme Dynamické Webové Stránky. Praha, Softpress 2003. ISBN 80-86497-40-2

[3] Andrei Alexandrescu, Moderní programování v C++, Praha, Computer Press 2004. ISBN 80-251-0370-6

[4] Matt Welsh, Matthias Kalle Dalheimer, Terry Dawson, Lar KaufmanPoužíváme Linux. Praha, Computer Press 2003 ISBN 80-7226-698-5

[5] A. Koenig, Rozumíme C++, Computer Press, 2003, ISBN 80-7226-656-X

[6] Nicolai M. Josuttis C++ Standardní knihovna a STL. Praha, Computer Press 2005, ISBN 80-251-0511-1

[7] Kolektiv autorů, PHP Programuje profesionálně, Praha , Computer Press 2004 ISBN 80-7226-310-2

[8] Refaktoring - Zlepšení existujícího kódu, Praha, Grada Publishing,2004, ISBN 80-247-0299-1

[9] Pixy: XSS and SQLI Scanner for PHP, http://pixybox.seclab.tuwien.ac.at/pixy/ [26.12.2007]

[10] PMD – PMD, http://pmd.sourceforge.net/ [26.12.2007]

[11] PHP Security Scanner project official website, http://securityscanner.lostfiles.de/ [30.12.2007]

[12] PHP-sat: PHP static analysis tool, http://www.program-transformation.org/PHP/PhpSat [2.1.2008]

[13] PHP-front: PHP4/5 syntax definition, parser, and pretty-printer, http://www.program-transformation.org/PHP/PhpFront [2.1.2008]

[14] PHC -- the open source PHP compiler, http://www.phpcompiler.org/ [2.5.2008]

[15] Stratego Program Transformation Language, http://www.program-transformation.org/Stratego/ [2.5.2008]

[16] Maketa, http://www.maketea.org/ [2.5.2008]

Přílohy

Příloha číslo 1 – dotazník pro programátory