NÁSTROJE NA ZVYŠOVÁNÍ KVALITY KÓDU

Jarmila Pavlíčková
Luboš Pavlíček

Katedra informačních technologií, VŠE v Praze, nám. W. Churchilla 4, 130 00 Praha 3, ČR
Email: pavjar@vse.cz, pavlicek@vse.cz

Abstrakt
Kvalitní software by měl být cílem všech programátorů. Existují softwarové nástroje, které se snaží odhalovat chyby v kódu či zlepšovat čitelnost a znovupoužitelnost kódu. Tyto nástroje jsou nezávislé na typu aplikace a snaží se vylepšovat kód za hranicí možností překladačů. Zaměřují se hlavně do následujících oblastí:

- formátování textu zdrojového kódu,
- kontrola konvencí - konvence pro názvy proměnných, konvence pro dokumentaci, ...
- hledání konstrukcí v programu, které často vedou k chybě - např. switch bez větve default,
- hledání chybných konstrukcí v programu - např. opomenutí synchronizace datového atributu, který je použit ve vláknu.

V příspěvku budou popsány možnosti některých volně dostupných nástrojů, které jsou k dispozici pro programátory v Javě: Jalopy, JRefactory, CheckStyle, PMD, FindBugs, JLint, JDepend, Pasta, JavaNCSS.

1. Úvod

Při tvorbě softwaru se mohou programátoři dopustit mnoha různých chyb, některé nemusejí znamenat jen špatnou funkčnost, mohou to být i chyby (mohl by být použit i výraz nedostatky), které vedou k problémům při implementaci změn, omezují znovupoužitelnost kódu apod.

Abychom lépe objasnili jaké chyby máme na mysli, uvádíme příklad. Jedná se o jednoduchý kód napsaný v Javě, metoda toUpperCase slouží k převodu malého písmena na velké, metoda main slouží jako první velmi jednoduchý test.

```
....
public static void main(String[] args){
      System.out.println(toUpperCase('q'));
}

private static char toUpperCase(char main){
      return (char) (main - 31);
}
....
```

Na tomto kódu lze ukázat několik typů chyb:

a) chyba v kódu – metoda toUpperCase pro písmeno q vrátí velké R. Tuto chybu lze snadno odhalit spuštěním programu – metodu main lze považovat za test funkčnosti metody toUpperCase,
b) skrytá chyba – pokud někdo na vstupu zadá číslici či velké písmeno, vrátí metoda nesmyslnou hodnotu, tuto chybu je možno odhalit komplexnějším testováním.

c) použití slova main pro identifikátor parametru je zavádějící – nevyjadřuje obsah položky a navíc je zavádějící vzhledem k existenci metody se stejným jménem. Vhodnější by bylo `private static char toUpperCase(char lower)`
d) nejvhodnější je nepsat vlastní metodu na převod na velká písmena a použít volání metody ze standardní knihovny Javy: `System.out.println (Character.toUpperCase('q'))`

Obecněji jsme pro potřeby našeho článku rozdělili možné chyby do následujících skupin:

a) aktuální chyby v kódu – odhalí se pomocí testů funkčnosti,
b) skryté/latentní chyby – chyby které jsou v kódu a které neodhalí aktuální testy funkčnosti tříd, příkladem může být chyba roku 2000,
c) chyby omezující čitelnost a pochopitelnost kódu – chybějící komentáře, názvy proměnných, které nevyjadřují obsah, atd.,
d) chyby vyplývající z nesprávné organizace kódu, omezující znovupoužitelnost kódu nebo znesnadňující refactoring či optimalizaci – zbytečné proměnné, nepoužívané metody, zbytečné vazby na další kód, nevyužití standardních knihoven atd.

Odhalování jednotlivých chyb je obtížné a neexistuje žádný spolehlivý způsob/postup, jak na všechny přijít. Existuje mnoho různých nástrojů, které se o to v různé míře pokoušejí. Tento článek se zaměřuje na stručný popis některých nástrojů vhodných pro programátory v Javě.
Samostatnou skupinu tvoří nástroje pro vytváření testů funkčnosti modulů a tříd a jejich používání. Příkladem takovéhoto nástroje může být JUnit[14] pro testování tříd. Této skupině nástrojů se však věnovat nebudeme.

Nástroje na automatické odhalování chyb jsme pro potřeby toho článku rozdělili do následujících skupin:

- formátování textu zdrojového kódu,
- kontrola konvencí pro psaní kódu,
- hledání problematických konstrukcí a skrytých chyb v programu,
- metriky kvality kódu,

2. Formátování textu zdrojového kódu

Cílem této skupiny nástrojů je naformátovat zdrojový kód tak, aby byl dobře čitelný a přehledný.

Základní funkčnost těchto nástrojů:

- formátování umístění závorek,
- odsazování bloků,
- doplňování/rušení mezer,
- lámání řádek,
- oddělování částí kódů prázdnými řádky,
- generování základů dokumentace pro javadoc,
- používání template pro hlavičky,
- export „obarvené“ verze do html s možností následného tisku,

Tyto nástroje umožňují si nadefinovat vlastní pravidla (o kolik se odsazuje.způsob formátování bloků, ...).

Velký význam mají tyto nástroje při analýze/změnách staršího kódu (refactoring) – naformátování pomůže lepšímu pochopení původního kódu. Proto jsou tyto funkce často obsaženy v nástrojích pro refactoring.

Formátování zdrojového kódu je problematické v situaci, kdy vytváříme další verzi kódu a současně používáme nějaký systém na správu verzí. Po přeformátování kódu systémy pro správu verzí nerozliší změny formátování od změn v kódu a označí celý kód za nový. Formátování zdrojového kódu by se tudíž mělo používat na začátku vývoje či na začátku analýzy staršího kódu, před uložením první verze do systému na správu verzí.

Z Open Source programů pro Javu si zaslouží zmínku dva:
- jalopy (jalopy.sourceforge.net),
- jrefactory (jrefactory.sourceforge.net) – součástí tohoto Open Source nástroje pro refactoring je modul PrettyPrinter pro formátování zdrojového textu kódu.

Mnoho z funkčnosti těchto nástrojů je obsaženo v současných prostředích pro vývoj aplikací:
- automatické odsazování kódu,
- zvýrazňování syntaxe,
- používání templatů pro jednotlivé třídy.

3. Kontrola konvencí pro psaní kódu

Tato skupina nástrojů provádí analýzu zdrojového textu programu a snaží se detekovat problémy omezující čitelnost a pochopitelnost kódu. Kritéria pro toto posouzení se obvykle odvozují z konvencí firmy Sun pro psaní zdrojového kódu (java.sun.com/docs/codeconv), většina nástrojů umožňuje definovat i vlastní konvence.

Typickým představitelem této skupiny nástrojů je program CheckStyle (checkstyle.sourceforge.net), jehož kontroly lze rozdělit do následujících skupin:
- kontrola existence a úplnosti komentářů pro Javadoc,
- kontrola konvencí pro pojmenování jednotlivých prvků kódu,
- kontrola speciálních, uživatelsky definovaných hlaviček (např. s autorskými právy),
- kontrola direktiv import (zda neobsahuje hvězdičky, zda neobsahuje nevhodné třídy – např. použití tříd z balíčků sun.* omezuje přenositelnost),
- kontroly velikosti kódu (délka metod, délka řádků, max. počet parametrů, ...),
- kontrola umístění mezer, kontrola odsazení,
- kontroly na používání složených závorek (např. za if vždy používat složené závorky i v případě, že obsahují pouze jeden příkaz), kontrola zarovnání složených závorek,
- kontrola některých nevhodných konstrukcí.

Program CheckStyle provádí i některé kontroly, které patří do následující skupiny (např. deklarace metody equals, která nepřekrývá metodu equals ze třídy Object či kontrola, zda při překrytí metody equals je překryta i metoda hashCode, kontroly na deklaraci a předávání výjimek apod.).

4. Hledání problematických konstrukcí v programu

Nástroje z této skupiny se soustředí na problémy omezující čitelnost kódu (podobně jako předchozí skupina) a na problémy se znovupoužitelností a kvalitou kódu. Částečně se překrývají s předchozí a následující skupinou.

Typickým představitelem této skupiny je program PMD (pmd.sourceforge.net). PMD pracuje se zdrojovým kódem podobným způsobem, jako překladač – provede syntaktickou analýzu kódu a převede ho do Abstract Syntax Tree nad kterým vyhledává problematické konstrukce. Součástí programu PMD je i modul CPD, který slouží k vyhledávání duplicitního kódu.

Chyby odhalené pomocí PMD lze rozdělit do následujících skupin:

- prázdné bloky catch, if, finaly,
- chyby v pojmenovávání prvků kódu – rozpor s konvencemi pro pojmenovávání prvků či příliš dlouhá či příliš krátká jména,
- kontrola používání složených závorek,
- kontroly na délky kódu – příliš dlouhý kód či příliš složitý kód (Cyclomatic complexity) omezují možnost porozumění kódu,
- vyhledávání nepoužívaného kódu – nepoužité proměnné, metody či parametry metod znamenají obtížnější čitelnost i obtížnější úpravy kódu,
- chyby v návrhu kódu – velké množství příkazů import je příznakem velké závislosti na okolí (malé zapouzdření), velké množství datových atributů či metod je známkou přílišné složitosti, throws na výjimku Exception a ne na konkrétní výjimky, používání stejných řetězců na mnoha místech v kódu – vhodnější je nadefinovat konstanty, vytváření jednotlivých instancí třídy Boolean (vhodnější je používat konstanty pro true a false ze třídy Boolean),
- nevhodné konstrukce – new String, porovnávání booleovských výrazů s hodnotami true či false, podmínky if, které obsahují pouze return true a return false, chybějící default u příkazu switch,

Některé z chyb, které vypíše program PMD je potřeba brát jako doporučení – např. pravidlo definující max. 12 znaků pro název proměnné je někdy příliš omezující.
Konkrétní představu o možnostech programu PMD si lze udělat z následující tabulky, která obsahuje výsledky testování dvou projektů – aktuální verze populárního webového kontejneru Tomcat[11] a objektové databáze Ozone[13].

Tabulka 1: Výsledky testování programem PMD

skupina chyb	**chyba**	**Tomcat 5.0.19**	**Ozone 1.1**
Basic	Avoid empty catch blocks	104	11
Basic	Avoid empty finally blocks	0	2
Basic	Avoid empty 'if' statements	16	16
Basic	Double checked locking is not thread safe in Java.	0	1
Basic	Ensure you override both equals() and hashCode()	1	13
Braces	Avoid using 'for' statements without curly braces	62	3
Braces	Avoid using if statements without curly braces	1877	18
Braces	Avoid using 'if...else' statements without curly braces	191	3
Braces	Avoid using 'while' statements without curly braces	18	2
Code Size	Avoid really long Classes.	28	0
Code Size	Avoid really long methods.	59	5
Code Size	The class has a Cyclomatic Complexity	87	28
Code Size	The constructor has a Cyclomatic Complexity	1	0
Code Size	The method has a Cyclomatic Complexity	170	39
Design	A high number of imports can indicate a high degree of coupling within an object.	21	0
Design	A high number of public methods and attributes in an object can indicate the class may need to be broken up for exhaustive testing may prove difficult.	17	0

Tabulka 1

skupina chyb	chyba	Tomcat 5.0.19	Ozone 1.1
Design	A high ratio of statements to labels in a switch statement. Consider refactoring.	3	1
Design	Avoid calls to overridable methods during construction	37	12
Design	Avoid instantiating Boolean objects; you can usually invoke Boolean.valueOf() instead.	42	3
Design	Avoid unnecessary comparisons in boolean expressions	2	9
Design	Avoid unnecessary if..then..else statements when returning a boolean	8	0
Design	Avoid using implementation types like 'ArrayList'; use the interface instead	30	1
Design	Avoid using implementation types like 'HashMap'; use the interface instead	63	2
Design	Avoid using implementation types like 'HashSet'; use the interface instead	1	2
Design	Avoid using implementation types like 'TreeMap'; use the interface instead	0	1
Design	Avoid using implementation types like 'Vector'; use the interface instead	14	8
Design	Deeply nested if..then statements are hard to read	15	9
Design	High amount of coupling within the class	38	8
Design	Switch statements should have a default label	15	3
Design	The catch clause shouldn't check the exception type - catch several exceptions instead	3	5
Design	This final field could be made static	9	0
Design	This for loop could be simplified to a while loop	0	1
Exception	A catch statement should never catch throwable since it includes errors	202	10
Exception	A signature (constructor or method) shouldn't have Exception in throws declaration	249	433
Import	Avoid importing anything from the package 'java.lang'	0	7
Import	No need to import a type that's in the same package	4	1
Naming	Avoid excessively long variable names	471	125
Naming	Avoid using short method names	0	10
Naming	Avoid variables with short names	657	758
Naming	Class names should begin with an uppercase character and not include underscores	1	0
Naming	Method name does not begin with a lower case character.	5	3
Naming	Method names should not contain underscores	0	25
Naming	Variables should start with a lowercase character	13	12
Naming	Variables that are final should be in all caps.	159	121
Naming	Variables that are not final should not contain underscores.	18	14
String	Avoid calling toString() on String objects; this is unnecessary	4	4
String	Avoid instantiating String objects; this is usually unnecessary.	12	15
String	The same String literal appears min 4 times in file	187	49
Unused Code	Avoid unused formal parameters	3	2
Unused Code	Avoid unused imports	0	36
Unused Code	Avoid unused local variables	76	33
Unused Code	Avoid unused private fields	17	6
Unused Code	Avoid unused private methods	9	3

5. Hledání skrytých chyb v programu

Tato skupina nástrojů prochází bytecode a hledá v něm konstrukce, které jsou obvyklé u skrytých chyb. I v tomto případě nelze zajistit 100% přesnost odhalení – programátor musí vždy posoudit, zda v daném případě je to opravdu chyba či pouze falešné upozornění.

V této kategorii si popíšeme dva nástroje: FindBugs a Jlint.

Nejstarší z nástrojů uvedených v tomto článku je program Jlint (artho.com/jlint/). Nejsilnější stránka tohoto programu je odhalování chyb v souvislosti s používání vláken (chyby v synchronizaci, nevhodné lockování proměnných, ...). Bohužel tento program se již delší dobu nevyvíjí.

Program FindBugs (www.cs.umd.edu/~pugh/java/bugs) je slibný projekt, který též analyzuje bytecode. V současné době již obsahuje většinu testů obsažených v programu Jlint a některé další. Přehled možností tohoto programu je vidět z následující tabulky, ve které jsou vidět výsledky testování programů Tomcat verze 5.0.19 a objektové databáze Ozone verze 1.1.

Tabulka 2: Výsledky testů programu FindBugs

	Tomcat 5.0.19	**Ozone 1.1**
chyby z hlediska bezpečnosti kódu		
Finalizers are public		3
Method returning array may expose internal representation	33	
chyby z hlediska výkonnosti		
Dubious method used	32	72
Empty finalizer		3
Redundant comparison to null	26	12
Should be a static inner class?	7	1
Unread field	11	3
Unwritten field		22
Useles control flow	4	5
skryté chyby		
Dropped or ignored exception	8	4
Equal objects must have equal hashcodes	1	18
Checking String equality using == or !=	5	
Incorrect definition of Iterator		2
Incorrect definition of Serializable class	5	33
Incorrect use of finalizers		3
Method ignores results of InputStream.read()		1
Null pointer dereference	2	2
Stream not closed on all paths	5	
chyby v používání vláken		
Inconsistent synchronization	31	43
Method invokes run()		2
Mutable static field	45	19
Naked notify method		1
Possible double check of field	3	2
Unsynchronized get method, synchronized set method	6	3
Using notify() rather than notifyAll() in method	2	2

6. Metriky kvality kódu

Programy této skupiny nemají za cíl zjišťovat nějaké nedostatky, ale vytvářet statistiky a metriky pro vytvořený kód, které lze použít při porovnávání složitosti a závislostí jednotlivých částí kódu a pro sledování vývojových trendů v kódu.
Jednoduché statistiky vytváří program JavaNCSS (www.kclee.com/clemens/java/javancss/), který počítá řádky zdrojového kódu a množství komentářů. Výstup může mít různé úrovně podrobnosti, následují globální výstupy za námi testované aplikace – Tomcat verze 5.0.19 a Ozone 1.1:

Tabulka 3: Výsledky programu JavaNCSS pro Tomcat 5.0.19:

Packages	Classes	Functions	NCSS	Javadocs	per
26.00	364.00	5268.00	42075.00	4515.00	Project
	14.00	202.62	1618.27	173.65	Package
		14.47	115.59	12.40	Class
			7.99	0.86	Function

Tabulka 4: Výsledky programu JavaNCSS pro Ozone 1.1:

Packages	Classes	Functions	NCSS	Javadocs	per
27.00	288.00	2445.00	16371.00	1215.00	Project
	10.67	90.56	606.33	45.00	Package
		8.49	56.84	4.22	Class
			6.70	0.50	Function

Program JDepend počítá metriky ze zdrojového kódu s cílem porovnat nezávislosti jednotlivých balíčků a tříd – jak je třída/balíček závislá na jiných, kolik tříd je závislých na konkrétní třídě, zda existuje cyklická závislost mezi třídami.

Zajímavým programem je i volně dostupný program Pasta od firmy Compuware (javacentral.compuware.com/pasta/index.htm), který vedle vytváření metrik umí nakreslit závislosti mezi třídami a balíčky a uložit je do souborů PNG. Také odhaluje cyklické závislosti.

7. Závěr

Efekt těchto nástrojů se zvýší při pravidelném používání, tj. při správném začlenění do vývojového procesu. Jsou dvě místa na kterých se uplatní:

- začlenění do vývojového prostředí programátora, většina výše uvedených nástrojů může být začleněna do různých vývojových prostředí (NetBeans, JBuilder ...), nejčastější je podpora prostředí Eclipse. Programátor má „online“ odezvu - po úspěšném přeložení třídy se ihned spouštějí tyto testy.
- začlenění do testovacího cyklu vytvářeného projektu. Většina nástrojů má pluginy do programu Ant[12], který se obvykle používá pro sestavování výsledného projektu. Tyto testy se poté pravidelně provádějí současně s testy funkčnosti.

Není překvapivé, že tyto nástroje často používají týmy pracující dle metodiky extrémního programování. Dle průzkumů je také často používají týmy vyvíjející Open Source programové vybavení.

Literatura:

1. http://jalopy.sourceforge.net
2. http://jrefactory.sourceforge.net
3. http://java.sun.com/docs/codeconv
4. http://checkstyle.sourceforge.net
5. http://pmd.sourceforge.net
6. http://artho.com/jlint
7. http://www.cs.umd.edu/~pugh/java/bugs
8. http://www.kclee.com/clemens/java/javancss
9. http://javacentral.compuware.com/pasta/index.htm
10. http://www.reasoning.com
11. http://jakarta.apache.org/tomcat
12. http://ant.apache.org/
13. http://www.ozone-db.org
14. http://www.junit.org