Java CodeCount™

Counting Standard

University of Southern California

Center for Systems and Software Engineering

June , 2007

Revision Sheet

Date	Version	Revision Description	Author
6/22/2007	1.0	Original Release	CSSE
1/2/2013	1.1	Updated document template	CSSE
1/14/2013	1.2	Added cyclomatic complexity	CSSE
11/21/2017	1.3	Added Halstead Volume	CSSE

Table of Contents

1. Definitions

1.1. **SLOC –** Source Lines of Code is a unit used to measure the size of software program. SLOC counts the program source code based on a certain set of rules. SLOC is a key input for estimating project effort and is also used to calculate productivity and other measurements.

1.2. **Physical SLOC –** One physical SLOC is corresponding to one line starting with the first character and ending by a carriage return or an end-of-file marker of the same line, and which excludes the blank and comment line.

1.3. **Logical SLOC –** Lines of code intended to measure "statements", which normally terminate by a semicolon (C/C++, Java, C#) or a carriage return (VB, Assembly), etc. Logical SLOC are not sensitive to format and style conventions, but they are language-dependent.

1.4. **Data declaration line or data line –** A line that contains declaration of data and used by an assembler or compiler to interpret other elements of the program.

The following table lists the Java keywords that denote data declaration lines:

abstract	boolean	const	int	long
byte	short	char	extends	float
double	implements	class	interface	native
void	static	package	private	public
protected	operator	volatile	template	

Table 1 Data Declaration Types

1.5. **Compiler Directives –** A statement that tells the compiler how to compile a program, but not what to compile.

The following table lists the Java keywords that denote compiler directive lines:

package	import

Table 2 Compiler Directives

1.6. **Blank Line –** A physical line of code, which contains any number of white space characters (spaces, tabs, form feed, carriage return, line feed, or their derivatives).

1.7. **Comment Line –** A comment is defined as a string of zero or more characters that follow language-specific comment delimiter.

Java comment delimiters are "//" and "/*". A whole comment line may span one line and does not contain any compliable source code. An embedded comment can co-exist with compliable source code on the same physical line. Banners and empty comments are treated as types of comments.

1.8. **Executable Line of code –** A line that contains software instruction executed during runtime and on which a breakpoint can be set in a debugging tool. An instruction can be stated in a simple or compound form.

- An executable line of code may contain the following program control statements:
  - Selection statements (if, ? operator, switch)
  - Iteration statements (for, while, do-while)
  - Empty statements (one or more ";")
  - Jump statements (return, goto, break, continue, exit function)
  - Expression statements (function calls, assignment statements, operations, etc.)
  - Block statements
- An executable line of code may not contain the following statements:
  - Compiler directives
  - Data declaration (data) lines
  - Whole line comments, including empty comments and banners
  - Blank lines

2. Checklist for source statement counts

PHYSICAL SLOC COUNTING RULES			
MEASUREMENT UNIT	ORDER OF PRECEDENCE	PHYSICAL SLOC	COMMENTS
Executable Lines	1	One Per line	Defined in 1.8
Non-executable Lines			
Declaration (Data) lines	2	One per line	Defined in 1.4
Compiler Directives	3	One per line	Defined in 1.5
Comments			Defined in 1.7
On their own lines	4	Not Included (NI)	
Embedded	5	NI	
Banners	6	NI	
Empty Comments	7	NI	
Blank Lines	8	NI	Defined in 1.6

LOGICAL SLOC COUNTING RULES				
NO.	STRUCTURE	ORDER OF PRECEDENCE	LOGICAL SLOC RULES	COMMENTS
R01	"for", "while", "foreach" or "if" statement	1	Count Once	"while" is an independent statement.
R02	*do {…} while (…); statement*	2	Count Once	Braces {…} and semicolon ; used with this statement are not counted.
R03	Statements ending by a semicolon	3	Count once per statement, including empty statement	Semicolons within "for" statement are not counted. Semicolons used with R01 and R02 are not counted.
R04	Block delimiters, braces {…}	4	Count once per pair of braces {..}, except where a closing brace is followed by a semicolon, i.e. };or an opening brace comes after a keyword "else".	Braces used with R01 and R02 are not counted. Function definition is counted once since it is followed by {…}.
R05	Compiler Directive	5	Count once per directive	

3. Examples

EXECUTABLE LINES

SELECTION Statement

ESS1 – if, else if, else and nested if statements

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
if (<boolean expression>) <statements>;	if (x != 0) System.out.print (“non-zero”);	1 1

if (<boolean expression>)
<statements>;
else <statements>;
	if (x > 0)
	System.out.print (“positive”);
	else
	System.out.print (“negative”);
		1
		1
		0
		1

if (<boolean expression>)
<statements>;
else if (<boolean expression>)
<statements>;.
.
.
else <statements>;
	if (x == 0)
	System.out.print (“zero”);
	else if (x > 0)
	System.out.print (“positive”);
	else {
	System.out.print (“negative”);
	}
		1
		1
		1
		1
		0
		1
		0

NOTE: complexity is not considered, i.e. multiple “&&” or “||” as part of the expression.	if ((x != 0) && (x > 0)) System.out.print (x);	1 1

ESS2 – ? operator

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
Exp1?Exp2:Exp3	x > 0 ? System.out.print (“positive”) : System.out.print (“negative”);	1

ESS3 – switch and nested switch statements

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

switch (<expression>)
{
case <constant 1> :
<statements>;
break;
default
<statements>;
}
	switch (number)
	{
	case 1:
	foo1();
	break;
	default
	System.out.print (“invalid case”);
	}
		1
		0
		0
		1
		1
		0
		1
		0

ESS4 – try-catch

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

try {} catch() {}	try {
	inputFileName=args[0];
	}
	catch (IOException e) {
	System.err.println(e);
	System.exit(1);
	}
		1
		1
		0
		1
		1
		1
		0

ITERATION Statement

EIS1 – for

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

for (initialization; condition; increment)
statement;
 	
NOTE: “for” statement counts as one, no matter how many optional expressions it contains, i.e.
for (i = 0, j = 0; I < 5, j < 10; i++, ,j++)
	for (i = 0; i < 10; i++)
	System.out.print (i);
		1
		1

EIS2 – empty statements (could be used for time delays)

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
for (i = 0; i < SOME_VALUE; i++) ;	for (i = 0; i < 10; i++) ;	2

EIS3 – while

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

while (<boolean expression>)
<statement>;
	while (i < 10)
	{
	System.out.print (i);
	i++;
	}
		1
		0
		1
		1
		0

EIS4 – do-while

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

do
{
<statements>;
} while (<boolean expression>);
	do
	{
	ch = getCharacter();
	} while (ch != '\n');
		1
		0
		1
		1

EIS5 – for-each

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
for (String name: moreNames) System.out.println(name.charAt(0));	for (String n: Names) System.out.println(ncharAt(0));	1 1

JUMP Statement

EJS1 – return

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
return expression	If (i=0) return;	2

EJS2 – break

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
break;	if (i > 10) break;	2

EJS3 – exit function

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
void exit (int return_code);	if (x < 0) exit (1);	2

EJS4 – continue

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

continue;	while (!done)
	{
	ch = getchar();
	if (char == '\n')
	{
	done = true;
	continue;
	}
	}
		1
		0
		1
		1
		0
		1
		1
		0
		0

EXPRESSION Statement

EES1 – function call

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<function_name> (<parameters>);	read_file (name);	1

EES2 – assignment statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

<name> = <value>;	x = y;
	char name[6] = "file1";
	a = 1; b = 2; c = 3;
		1
		1
		3

EES3 – empty statement (is counted as it is considered to be a placeholder for something to call attention)

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
one or more ";" in succession	;	1 per each

BLOCK Statement

EBS1 – block=related statements treated as a unit

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

/* start of block */
{
<definitions>
<statement>
}
/* end of block */
	/* start of block */
	{
	i = 0;
	System.out.print ("%d", i);
	}
	/* end of block */
		0
		0
		1
		1
		1
		0

DECLARATION OR DATA LINES

DDL1 – function prototype, variable declaration

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<type> <name> (< parameter_list>);	Public static void foo (int param);	1
<type> <name>;	double amount;	1
Class<T>	Iterator<String>	1

COMPILER DIRECTIVES

CDL1 – directive types

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
package <package_name>;	package test;	1
import <package_name>;	import java.io*;	1

4. Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly independent paths through a program. It is measured for each function, procedure, or method according to each specific program language. This metric indicates the risk of program complexity and also determines the number of independent test required to verify program coverage.

The cyclomatic complexity is computed by counting the number of decisions plus one for the linear path. Decisions are determined by the number of conditional statements in a function. A function without any decisions would have a cyclomatic complexity of one. Each decision such as an if condition or a for loop adds one to the cyclomatic complexity.

The cyclomatic complexity metric v(G) was defined by Thomas McCabe. Several variations are commonly used but are not included in the UCC. The modified cyclomatic complexity counts select blocks as a single decision rather than counting each case. The strict or extended cyclomatic complexity includes boolean operators within conditional statements as additional decisions.

Cyclomatic Complexity	Risk Evaluation
1-10	A simple program, without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk program
> 50	Untestable program, very high risk

For Java, the following table lists the conditional keywords used to compute cyclomatic complexity.

Statement	CC Count	Rationale
if	+1	if adds a decision
else if	+1	else if adds a decision
else	0	Decision is at the if statement
switch case	+1 per case	Each case adds a decision – not the switch
switch default	0	Decision is at the case statements
for	+1	for adds a decision at loop start
while	+1	while adds a decision at loop start or at end of do loop
do	0	Decision is at while statement – no decision at unconditional loop
try	0	Decision is at catch statement
catch	+1	catch adds a decision
ternary ? :	+1	Ternary ? adds a decision – : is similar to default or else

5. Halstead Volume

Halstead Volume is one of several Halstead complexity measures introduced by Maurice Howard Halstead in 1977. Halstead observed that software metrics should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. Thus, Halstead Volume is computed statically from the code.

Halstead Volume can be computed using the following formula: -

$$N * \log_2(n)$$

Where N is the total Number of Operators + the total Number of Operands, n is the number of Unique Operators + the number of Unique Operands.

Halstead Volume can be defined as "[…] the size of the implementation of an algorithm. The computation […] is based on the number of operations performed and operands handled in the algorithm. Therefore [it] is less sensitive to code layout than the lines-of-code measures."

Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module. Halstead provides various indicators of the module's complexity.

Halstead metrics allow you to evaluate the testing time of any language source code for which we write the code. Halstead's goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (analogous to the gas equation). Thus his metrics are actually not just complexity metrics.

5.1) Computing Maintainability Index

Maintainability Index can be computed using the following equations: -

- 3-Metric Formula MI = 171 - 5.2 * ln(V) - 0.23 * (G) - 16.2 * ln(LOC)
- 4-Metric Formula: 171 - 5.2 * log2(V) - 0.23 * G - 16.2 * log2 (LOC) + 50 * sin (sqrt(2.4 * CM)

Where V is the average Halstead Volume per module, G is the average extended cyclomatic complexity per module, LOC is the average lines of code per module and CM is average percent of lines of comments per module.

For these calculations, we treat a function to be a module.

Where V is the average Halstead Volume per module, G is the average extended cyclomatic complexity per module and LOC is the average lines of code per module

5.2) List of Operators and Keywords

The following keywords and operators are counted towards Halstead Volume: -

i) Arithmetic Operators

Operator	Name	Example	Result
+	unary plus	+a	the value of a after promotions
-	unary minus	-a	the negative of a

+	addition	a + b	the addition of a and b
-	subtraction	a - b	the subtraction of b from a
*	product	a * b	the product of a and b
/	division	a / b	the division of a by b
%	modulo	a % b	the remainder of a divided by b
`	bitwise NOT	`a	the bitwise NOT of a
&	bitwise AND	a & b	the bitwise AND of a and b
|	bitwise OR	a | b	the bitwise OR of a and b
^	bitwise XOR	a ^ b	the bitwise XOR of a and b
<<	bitwise left shift	a << b	a left shifted by b
>>	bitwise right shift	a >> b	a right shifted by b
>>>	unsigned right shift	a >>> b	a unsigned right shifted by b

ii) Assignment Operators

Operator	Name	Example	Result	Equivalent of
=	basic assignment	a = b	a becomes equal to b	
+=	addition assignment	a += b	a becomes equal to the addition of a and b	a = a + b
-=	subtraction assignment	a -= b	a becomes equal to the subtraction of b from a	a = a – b
*=	multiplication assignment	a *= b	a becomes equal to the product of a and b	a = a * b
/=	division assignment	a /= b	a becomes equal to the division of a by b	a = a / b
%=	modulo assignment	a %= b	a becomes equal to the remainder of a divided by b	a = a % b
&=	bitwise AND assignment	a &= b	a becomes equal to the bitwise AND of a and b	a = a & b
|=	bitwise OR assignment	a |= b	a becomes equal to the bitwise OR of a and b	a = a | b
^=	bitwise XOR assignment	a ^= b	a becomes equal to the bitwise XOR of a and b	a = a ^ b
<<=	bitwise left shift assignment	a <<= b	a becomes equal to a left shifted by b	a = a << b
>>=	bitwise right shift assignment	a >>= b	a becomes equal to a right shifted by b	a = a >> b
>>>=	unsigned right shift			

iii) Comparison Operators

Operator	Name	Example	Result
==	equal to	a == b	a is equal to b
!=	not equal to	a != b	a is not equal to b
<	less than	a < b	a is less than b
>	greater than	a > b	a is greater than b
<=	less than or equal to	a <= b	a is less than or equal to b
>=	greater than or equal to	a >= b	a is greater than or equal to b

iv) Increment/Decrement Operator

Operator	Name	Example	Result
expr++	Postfix Increment	a++	use current value of a then a = a + 1
expr--	Postfix Decrement	a--	use current value of a then a = a - 1
++expr	Prefix Increment	++a	Do a = a + 1, then use value of a
--expr	Prefix Decrement	--a	Do a = a - 1, then use value of a

v) Logical Operators

Operator	Name	Example	Result
!	logical NOT	!a	the logical negation of a
&&	logical AND	a && b	the logical AND of a and b
||	logical OR	a || b	the logical OR of a and b

vi) Other Operators

Operator	Name	Example	Result
(...)	function call	f(...)	call the function f(), with zero or more arguments
(type)	type cast	(type)a	cast the type of a to type
instanceof	instanceof	a instanceof b	checks whether a is an object of b
? :	conditional operator	a ? b : c	if a is logically true (does not evaluate to zero) then evaluate expression b, otherwise evaluate expression c
{…}	block delimiters	{...}	Marks a block of code.

vii) Member Access Operators

Operator	Name	Example	Result
[…]	array subscript	a[b]	access the b^{th} element of array a
.	member access	a.b	access member b of a

viii) Keywords

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	for	final	finally
float	goto	if	implements
import	int	interface	long
native	new	package	private
protected	public	return	short

static	strictfp	super	switch
synchronized	this	throw	throws
transient	try	void	volatile
while			