



Measurement of Software Complexity with *Testwell CMT++* *Testwell CMTJava*

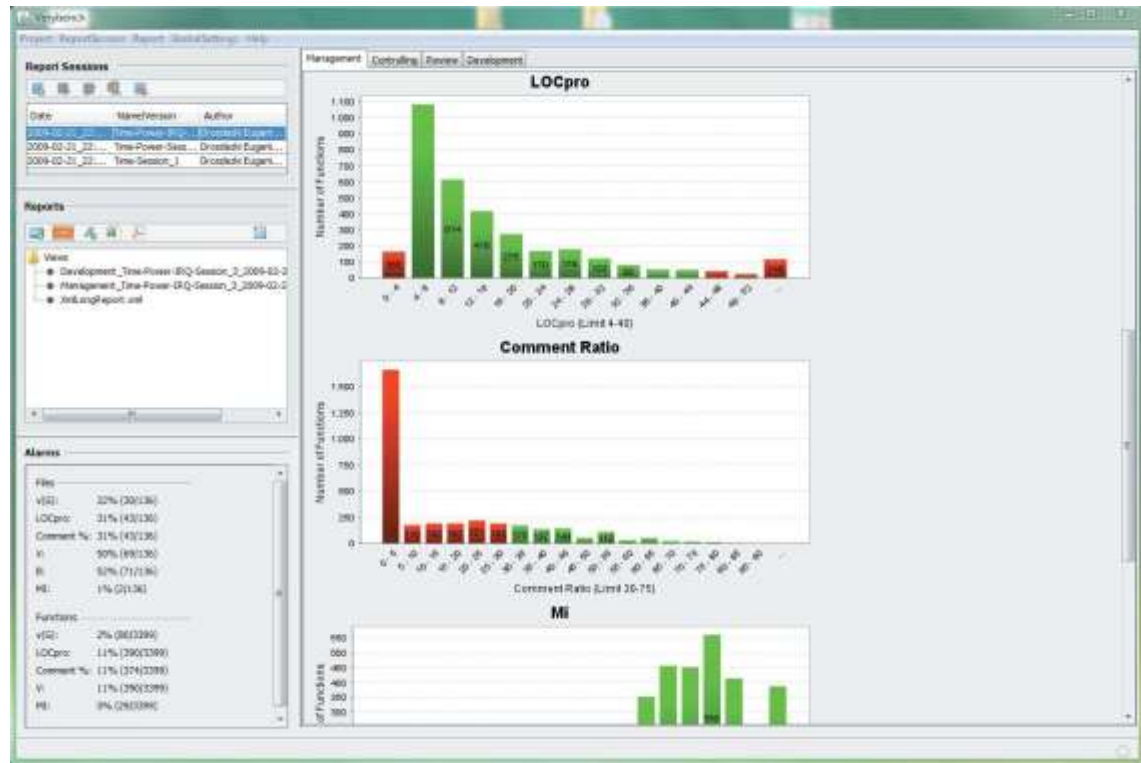
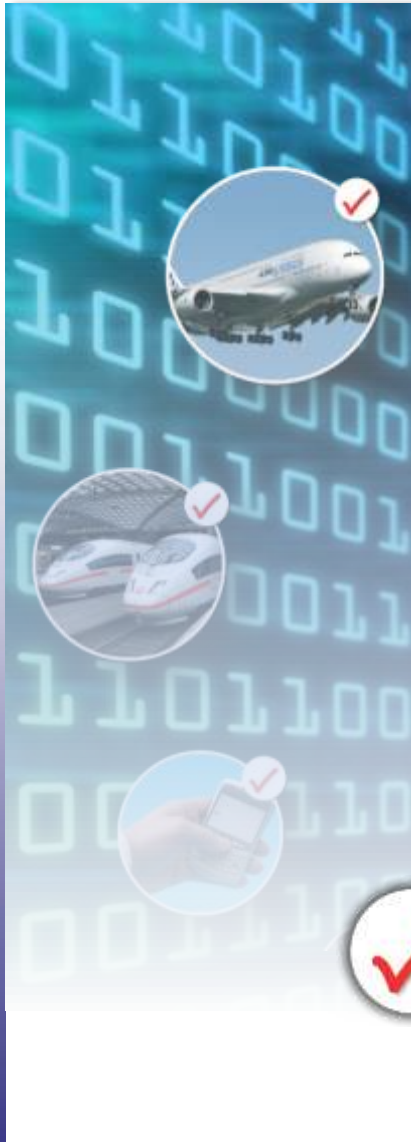




Code Complexity Measurements

Code Complexity Measurement Tools

Testwell CMT++ for C, C++ (and C#)
 Testwell CMTJava for Java

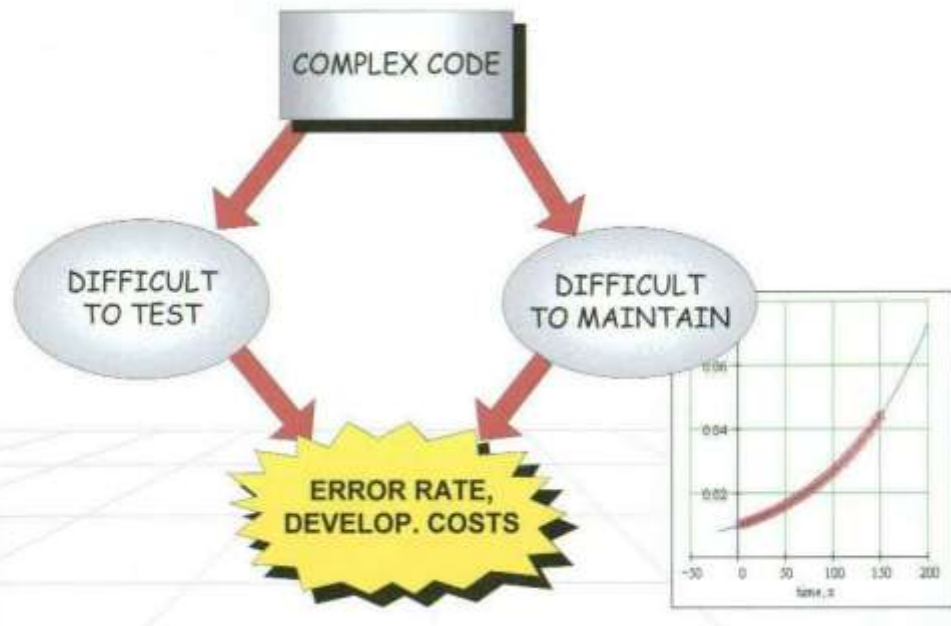




Code Complexity Measurements

Code complexity correlates with the defect rate and robustness of the application program

Wish to locate complex code



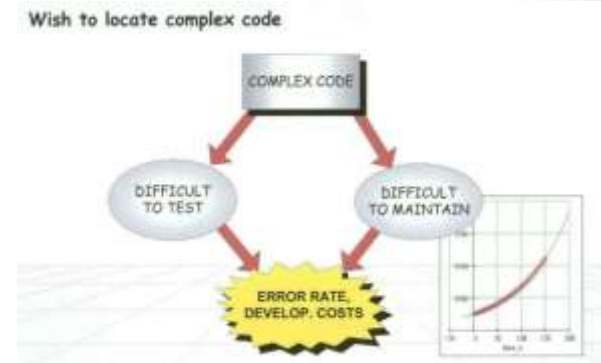


Code Complexity Measurements



Code with good complexity:

- ✓ contains less errors
- ✓ is easier and faster to test
- ✓ is easier to understand
- ✓ is easier to maintain



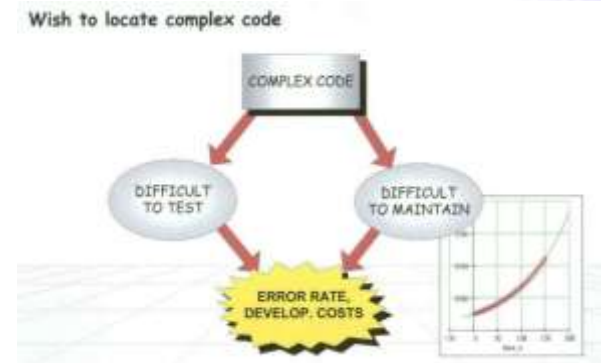


Code Complexity Measurements

Code complexity metrics are used to locate complex code

To obtain a high quality software with low cost of testing and maintenance, the code complexity should be measured as early as possible in coding.

→ developer can adapt his code when recommended values are exceeded.

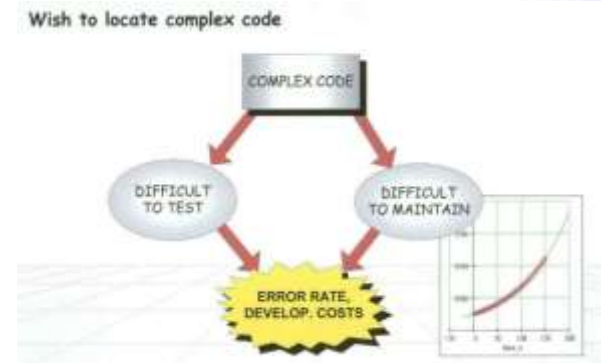




Code Complexity Measurements

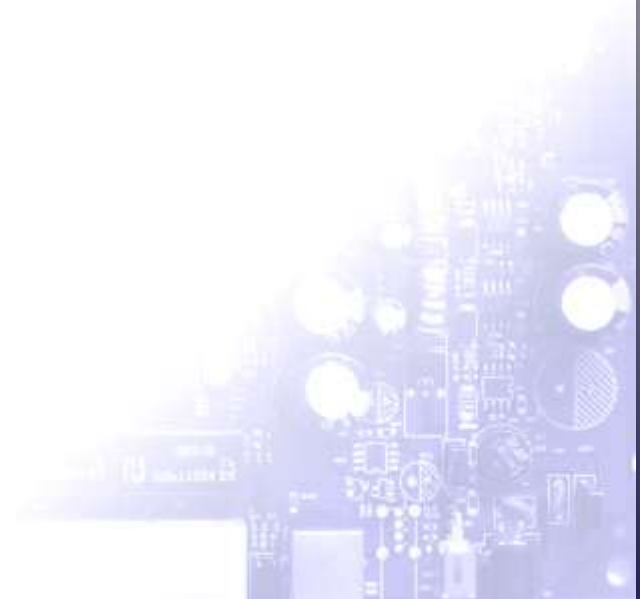
Metrics shown by Testwell CMT++ / CMTJava:

- ✓ Lines of Code metrics
- ✓ McCabe Cyclomatic number
- ✓ Halstead Metrics
- ✓ Maintainability Index





Lines of code metrics





Lines of code metrics



Most traditional measures used to quantify software complexity. They are simple, easy to count, and very easy to understand. They do not, however, take into account the intelligence content and the layout of the code.

Testwell CMT++ calculates the following lines-of-code metrics:

- **LOCphy**: number of physical lines
- **LOCbl**: number of blank lines (a blank line inside a comment block is considered to be a comment line)
- **LOCpro**: number of program lines (declarations, definitions, directives, and code)
- **LOCcom**: number of comment lines



Recommandations:

Function length should be 4 to 40 program lines.

- A function definition contains at least a prototype, one line of code, and a pair of braces, which makes 4 lines.
- A function longer than 40 program lines probably implements many functions. (*Exeption: Functions containing one selection statement with many branches*)

→ Decomposing them into smaler functions often decreases readability.



Recommandations:

File length should be 4 to 400 program lines.

- The smallest entity that may reasonably occupy a whole source file is a function, and the minimum length of a function is 4 lines.
- Files longer than 400 program lines (10..40 functions) are usually too long to be understood as a whole.

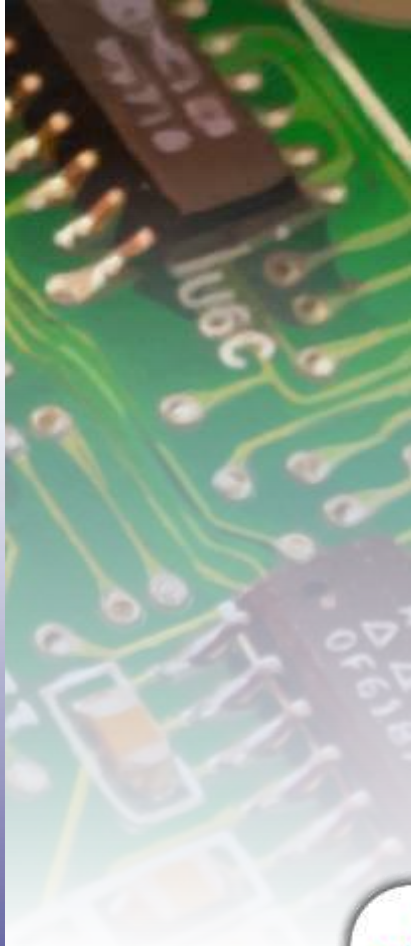


Recommandations:

Comments

- At least 30 % and at most 75 % of a file should be comments.
- If less than one third of a file is comments the file is either very trivial or poorly explained.
- If more than 75% of a file are comments, the file is not a program but a document.

(Exeption: In a well-documented header file percentage of comments may sometimes exceed 75%)



McCabe Cyclomatic Number



McCabe Cyclomatic Number

The cyclomatic complexity $v(G)$ has been introduced by Thomas McCabe in 1976.

Measures the number of linearly-independent paths through a program module (Control Flow).

The McCabe complexity is one of the more widely-accepted software metrics, it is intended to be independent of language and language format.

Considered as a broad measure of soundness and confidence for a program.





McCabe Cyclomatic Number

$v(G)$ is the number of conditional branches.
 $v(G) = 1$ for a program consisting of only sequential statements.

For a single function; $v(G)$ is one less than the number of conditional branching points in the function.

The greater the cyclomatic number is the more execution paths there are through the function, and the harder it is to understand.





How McCabe Metrics are calculated with CMT++:

Increase of McCabe cyclomatic number $v(G)$ by one:

- if-statement (introduces a new branch to the program)
- Iteration constructs such as for- and while-loops
- Each case ...: part in the switch-statement
- Each catch (...) part in a try-block
- Construction `expr1 ? expr2 : expr3`.





McCabe Cyclomatic Number

In CMT++ the branches generated by conditional compilation directives are also counted to $v(G)$.

(Even if conditional compilation directives do not add branches to the control flow of the executable program, they increase the complexity of the program file that the user sees and edits.)

$v(G)$ is insensitive to unconditional branches like goto-, return and break-statements although they surely increase complexity.





McCabe Cyclomatic Number

→ In summary, the following language constructs increase the cyclomatic number by one:

if (...)
case ...:
||
#ifdef

for (...)
catch (...)
?
#ifndef

while (...)
&&
#if
#elif





McCabe Cyclomatic Number

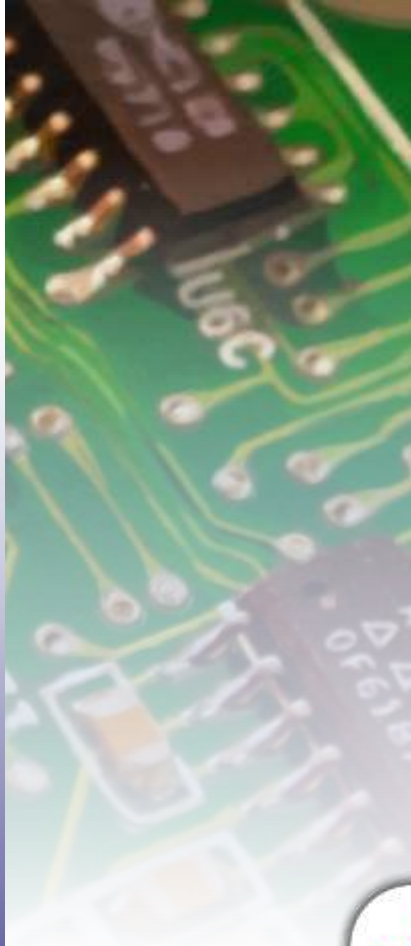
For dynamic testing, the cyclomatic number $v(G)$ is one of the most important complexity measures.

Because the cyclomatic number describes the control flow complexity, it is obvious that modules and functions having high cyclomatic number need more test cases than modules having a lower cyclomatic number.

Rule:

each function should have at least as many test cases as indicated by its cyclomatic number.





Recommandations:

- The cyclomatic number of a function should be less than 15.
If a function has a cyclomatic number of 15, there are at least 15 (but probably more) execution paths through it.
- More than 15 paths are hard to identify and test.
Functions containing one selection statement with many branches make up an exception.
- A reasonable upper limit Cyclomatic number of a file is 100.



Halstead Metrics





- Developed by Maurice Halstead (sen.)
 - Introduced 1977
 - Used and experimented extensively since that time
They are one of the oldest measures of program complexity
- Strong indicators of code complexity.
- Often used as a maintenance metric.





Halstead metrics



B	Estimated number of bugs
D	difficulty level, error proneness
E	effort to implement
L	program level
N	program length
N1	number of operators
N2	number of operands
n	vocabulary size (n_1+n_2)
n1	number of unique operators
n2	number of unique operands
T	implementation time / time to understand
V	volume: size of the implementation of an algorithm





Halstead's metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand.

Then is counted

- number of unique (distinct) operators (n_1)
- number of unique (distinct) operands (n_2)
- total number of operators (N_1)
- total number of operands (N_2)

All other Halstead measures are derived from these four quantities with certain fixed formulas as described later.





Operands:

IDENTIFIER

all identifiers that are not reserved words

TYPENAME

TYPESPEC (type specifiers)

Reserved words that specify

type: *bool, char, double, float, int, long, short, signed, unsigned, void*. This class also includes some compiler specific nonstandard keywords.

CONSTANT

Character, numeric or string constants.





Operators (1/2):

SCSPEC (storage class specifiers)

Reserved words that specify storage class: *auto, extern, inlin, register, static, typedef, virtual, mtuable*.

TYPE_QUAL (type qualifiers)

Reserved words that qualify type: *const, friend volatile*.

RESERVED

Other reserved words of C++: *asm, break, case, class, continue, default, delete, do, else, enum, for, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const_cast, static_cast, dynamic_cast, reinterpret_cast, typeid, template, explicit, true, false, typename*. This class also includes some compiler specific nonstandard keywords.





Halstead metrics

Operators (2/2):

OPERATOR

!	!=	%	%=	&	&&	
&=	()	*	*=	+	++	+=
,	-	--	-=	->
/	/=	:	::	<	<<	<<=
<=	=	==	>	>=	>>	>>=
?	[]	^	^=	{}		=
~ts						





Halstead metrics

The following control structures *case ...: for (...) if (...) switch (...) while for (...) and catch (...)* are treated in a special way.

The colon and the parentheses are considered to be a part of the constructs.

The case and the colon or the *for (...) if (...) switch (...) while for (...) and catch (...)* and the parentheses are counted together as one operator.





Program length (N)

The program length (N) is the sum of the total number of operators and operands in the program:

$$N = N1 + N2$$

Vocabulary size (n)

The vocabulary size (n) is the sum of the number of unique operators and operands:

$$n = n1 + n2$$





Program volume (V)

= information content of the program

It is calculated as the program length times the 2-base logarithm of the vocabulary size (n):

$$V = N * \log_2(n)$$

Halstead's volume (V) describes the size of the implementation of an algorithm.





Recommandations:

The volume of a **function** should be at least 20 and at most 1000.

- The volume of a parameterless one-line function that is not empty; is about 20.
- A volume greater than 1000 tells that the function probably does too many things.

The volume of a **file** should be at least 100 and at most 8000.

These limits are based on volumes measured for files whose LOCpro and v(G) are near their recommended limits.





Difficulty level (D)

The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program.

D is also proportional to the ration between the total number of operands and the number of unique operands.
(i.e. if the same operands are used many times in the program, it is more prone to errors)

$$D = (n1 / 2) * (N2 / n2)$$





Program level (L)

The program level (L) is the inverse of the error proneness of the program.

I.e. A low level program is more prone to errors than a high level program.

$$L = 1 / D$$





Effort to implement (E)

The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.

$$E = V * D$$





Time to implement (T)

The time to implement or understand a program (T) is proportional to the effort.

Halstead has found that dividing the effort by 18 give an approximation for the time in seconds.

$$T = E / 18$$





Number of delivered bugs (B)

The number of delivered bugs (B) correlates with the overall complexity of the software.

$$B = \frac{E^{2/3}}{3000}$$

Estimate for the number of errors in the implementation.
Delivered bugs in a file should be less than 2.

Experiences have shown that, when programming with C or C++, a source file almost always contains more errors than B suggests.





B is an important metric for dynamic testing:

The number of delivered bugs approximates the number of errors in a module.

As a goal at least that many errors should be found from the module in its testing.




Maintainability Index (MI)





Maintainability Index (MI)



Maintainability Index:
Calculated with certain formulae
from lines-of-code measures,
McCabe measure
and Halstead measures.

Indicates when it becomes cheaper and/or less risky to
rewrite the code instead to change it.





Two variants of Maintainability Index:

→ One that contains comments (MI) and one that does not contain comments (MIwoc).

Actually there are three measures:

- MIwoc: Maintainability Index without comments
- MIcw: Maintainability Index comment weight
- MI: Maintainability Index = MIwoc + MIcw





Maintainability Index (MI)

$$MI_{woc} = 171 - 5.2 * \ln(\text{aveV}) - 0.23 * \text{aveG} - 16.2 * \ln(\text{aveLOC})$$

- aveV = average Halstead Volume **V** per module
- aveG = average extended cyclomatic complexity **v(G)** per module
- aveLOC = average count of lines **LOCphy** per module
- perCM = average percent of lines of comments per Module





Maintainability Index (MI)

$$MI_{cw} = 50 * \sin(\sqrt{2,4 * perCM})$$

- perCM = average percent of lines of comments per module

$$MI = MI_{woc} + MI_{cw}$$





Maintainability Index (MI, with comments) values:

85 and more

Good maintainability

65-85

Moderate maintainability

< 65

Difficult to maintain

with really bad pieces of code (big, uncommented, unstructured) the MI value can be even negative





CMT++ Summary Report

file:///C:/Users/ed/cmt/CMHTML/index.html

Links anpassen Weitere Lesezeichen

```

.....
CMT++, Complexity Measures Tool for C/C++, Version 4.2
.....
COMPLEXITY MEASURES REPORT
.....
Copyright (c) 1993-2007 Testwell Oy
.....
License notice: This is a limited period evaluation copy license.

The input CMT++ report was produced at Mon Jan 12 17:11:08 2009
cmt options: -o C:\Users\ed\cmt\report.tmp -f C:\Users\ed\cmt\files.txt
Html'ized by cmt2html v2.2 at Mon Jan 12 17:11:08 2009
cmt2html options: -i C:\Users\ed\cmt\report.tmp

This is SUMMARY view. Go to DETAILED view. See instructions.

```

Alarms-1	Measured object	v(G)	LOCphy	LOCpro	c%	Y	B	MI
	calibrate.c	20	172	107	-	3195	1.24	110
	do_mounts.c	69	406	321	-	11535	3.18	105
	do_mounts_md.c	47	280	213	-	8280	2.39	88
	do_mounts_rd.c	46	429	313	-	11742	3.05	107
	msgutil.c	15	127	96	-	2751	1.02	95
	OVERALL (24 %)							

OVERALL SUMMARY:

Measure	5 Files			40 Functions		
	Alarmed	%	Limits	Alarmed	%	Limits
Cyclomatic number V(G)	0	0	1-100	4	10	1-15
Program lines LOCpro	0	0	4-400	12	30	4-40
Comment %	5	100	30-75	19	47	30-75
Volume V	3	60	100-8000	9	22	20-1000
Estimated number of bugs B	3	60	0-2	0	0	n/a
Maintainability index MI	0	0	65-	1	0	65-



Testwell CMT++/ CMTJava



```

report - Editor
Datei Bearbeiten Format Ansicht ?
-----
CMT++, Complexity Measures Tool for C/C++, Version 4.2
-----
COMPLEXITY MEASURES REPORT
-----
Copyright (c) 1993-2007 Testwell oy
-----

License notice: This is a limited period evaluation copy license.

This report was produced at Mon Jan 12 17:17:18 2009
Options: -o C:\Users\ed\cmt\report.txt -f C:\Users\ed\cmt\files.txt

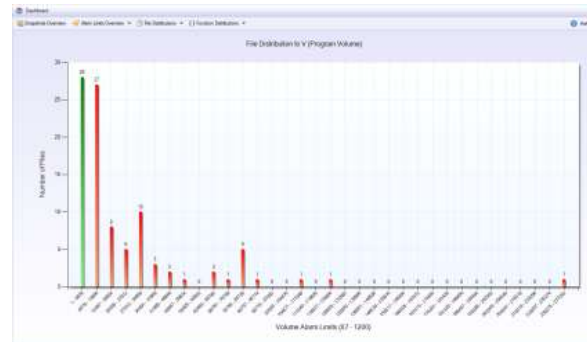
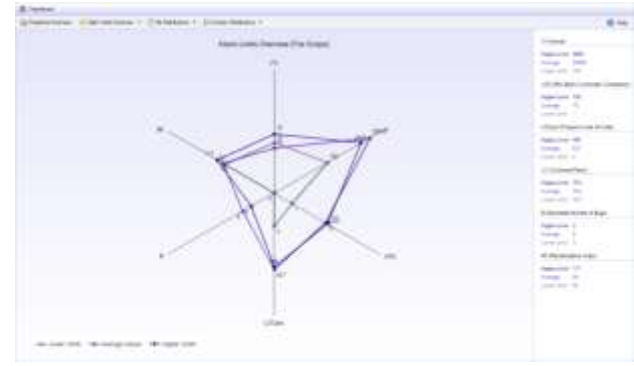
File: C:\Users\ed\Desktop\TestFiles\linux-2.6.26.8\linux-2.6.26.8\init\calibrate.c
-----
Line Measured object          v(G) LOCphy LOCpro  c%    v    B    MI
-----
13  lpj_setup()                 1      5      5      92  0.02  121
31  calibrate_delay_direct()     9     72     42~   1123- 0.39  101
104 calibrate_delay_direct()   1      1      1~    38  0.01  152
114 calibrate_delay()          10     59     47~   1146- 0.39   96
-----
172 calibrate.c                 20    172   107 -   3195  1.24  110
-----

File: C:\Users\ed\Desktop\TestFiles\linux-2.6.26.8\linux-2.6.26.8\init\do_mounts.c
-----
Line Measured object          v(G) LOCphy LOCpro  c%    v    B    MI
-----
31  load_randisk()                1      3      3     104  0.02  121
38  readonly()                   2      7      7      90  0.02  116
46  readwrite()                   2      7      7      96  0.02  115
57  name_to_dev_t()              22~    86    55~   1832- 0.57   90
144 root_dev_setup()            1      5      5      88  0.02  121
152 rootwait_setup()           2      7      7      88  0.02  116
163 root_data_setup()          1      5      5      61  0.01  123
170 fs_names_setup()           1      5      5      61  0.01  123
177 root_delay_setup()         1      5      5      92  0.02  121
187 get_fs_names()             7     26    24 -    615  0.19   83
214 do_mount_root()            3     14    13 -    520  0.12   95
229 mount_block_root()         9     52    44~   1127- 0.30   91
283 mount_nfs_root()           3     10     9 -    186  0.04  106
296 change_floppy()            9     27    27 -    889  0.22   82
325 mount_root()               9     28    26 -    447  0.07   98
354 prepare_namespace()        14     52    38 -    955  0.19   91
-----
406 do_mounts.c                 69    406   321 -  11535- 3.18-  105
-----

```



Verybench for Testwell CMT++/ CMTJava



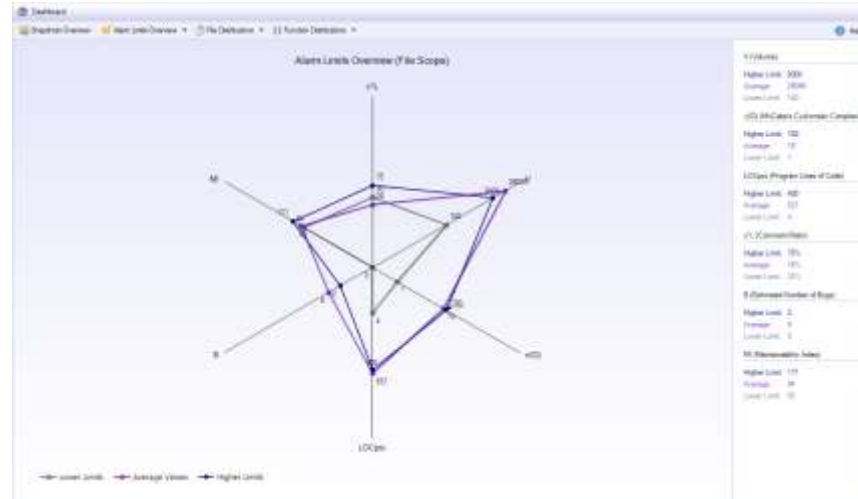
File Name	Function	Lines	LOCs	V	E	LOCs	W
main.cpp		25	100	100	100	100	100
main.h		10	20	20	20	20	20
main.o		10	20	20	20	20	20
main.exe		10	20	20	20	20	20
main.lib		10	20	20	20	20	20

Verybench:
Graphical front-end for Testwell Complexity Measurement Tools



Snapshots Overview

shows the course of the measured source code's quality over time by stating the alarm ratios of the latest six snapshots.

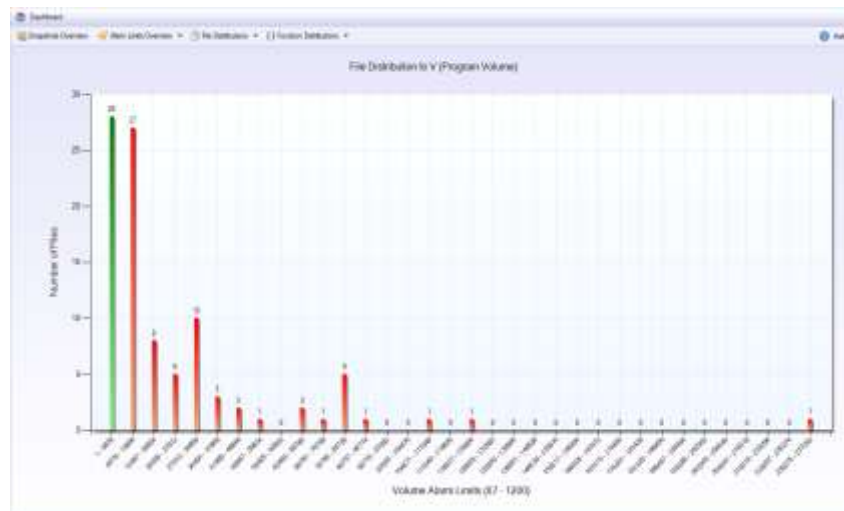


Alarm-Limits Overview

integrates all configurable Alarm Limits into a Radar Chart for each file and function.

Every axis represents a different alarming metric with its configurable lower and higher Alarm Limits.

The Radar Chart basically shows the deviation of a metric's current value from its lower and higher Alarm Limits.



Distribution of Metrics

shown on file and function levels for:

V (Program Volume)

c% (Comment Ratio)

LOCpro (Program Lines of Code)

v(G) (McCabe' Cyclomatic Number)

In addition B (Estimated Number of Bugs) is shown on file level



Metrics

File Scope Metrics (Snapshot: 6 Files) / Max. Alarms: 36

File Name	Functions	Alarms	a%	c%	LOCs	V	E	+SD	SB
bk-well-t.c	3	2	67%	33%	151	3462.76	3.882	36	127
bk-well-u.c	23	2	9%	8%	329	1000.00	2.34	29	106
bk-well-c.c	13	2	15%	15%	204	1000.00	2.34	29	122
beg.c	32	4	13%	13%	289	1000.00	8.425	39	99
display-well.c	34	5	15%	15%	344	1000.00	10.807	40	113
prod-well-c.c	11	5	45%	45%	105	1000.00	2.34	23	112

Source Code

```

1
2 * Functions related to safety, to completion
3
4 #include <sys/types.h>
5 #include <sys/time.h>
6 #include <sys/timeb.h>
7 #include <sys/stat.h>
8 #include <sys/errno.h>
9 #include <sys/unistd.h>
10 #include <sys/param.h>
11
12 #include "tbl.h"
13
14 #define DEFINE_PER_OP(tbl_head, bk_opu_bond)
15
16
17 * Setting action handler - move alarm to local list and keep over them
18 * while passing them to the alarm registered handler
19
20 #define void tbl_alarm_getting(tbl_alarm_action *a)
21
22 #define tbl_head *cpu_jar, local_jar

```

Metrics View

overview over all (both alarming and non-alarming) metrics of the measured files and functions



Thank You



Thank you for your time!

Your Verifysoft Team