CMTJava

Complexity Measures Tool for Java

User's Guide
Version 4.0

Testwell

CMTJava – Complexity Measures Tool for Java User's Guide, V4.0

March 2018 (upgraded from version 3.0 to 4.0)

Previous document versions:

September 2012 (upgraded from version 2.2 to 3.0)

December 2008 (upgraded from version 2.1 to 2.2)

September 2006 (upgraded from version 2.0 to 2.1)

September 2005 (upgraded from version 1.4 to 2.0)

January 2004 (upgraded from version 1.3 to 1.4)

August 2002 (upgraded from version 1.2 to 1.3)

April 2002 (upgraded from version 1.0 to version 1.2)

December 2001 (initial version, for version1.0)

Testwell Oy
Hermiankatu 8
FI-33720 Tampere
Finland

`http://www.testwell.fi`

Verifysoft Technology GmbH
In der Spöck 10-12
DE-77652 Offenburg
Germany

`http://www.verifysoft.de`

Contents

1. About This Guide

1.1. Overall

This guide is written for the CMTJava, Complexity Measures Tool for Java, version 4.0. We assume that you are acquainted with the Java language.

CMTJava is currently available on Windows, Linux, Solaris, macOS and HPUX. This guide is intended to be used in all of those environments and describes the basic functionality and command line based use of CMTJava.

The tool input is a set of Java source files. They are assumed to be syntactically correct, they do not otherwise need to be related to each other. The tool generates various types (depending of the used options) of reports: textual, HTML, XML, JSON, Excel.

On Windows platform, CMTJava can additionally be used via a graphical user interface (GUI). Effectively it is a graphical program layer for using the basic command line mode of tool components and for easily viewing the generated reports. The CMTJava GUI has its own on-line help and it is not described in this guide.

The examples of this guide have been worked up at the command-line prompt on a Windows machine. If your environment is something else, you should have no problems in understanding the examples and in transforming them to your environment, because effectively the only differences are in the syntax of file names.

This guide is organized as follows:

- chapter 2 “Introducing CMTJava” describes the properties and purpose of the system.
- chapter 3 “Installing CMTJava” describes the overall arrangements of all CMTJava installations.
- chapter 4 “Configuring CMTJava” describes how you can configure CMTJava and set ‘company standards’.
- chapter 5 “Using CMTJava” gives the operating instructions of CMTJava. An example of using CMTJava is presented.
- chapter 6 “Interpreting Complexity Measures” discusses the interpretation and use of the software metrics calculated by CMTJava.
- Appendix A “The Source Code Language” describes the way CMTJava analyzes Java source code.
- Appendix B “How the Measures Are Calculated” specifies how the individual complexity measures are calculated from the source code.

- The error messages of the cmtjava tool are listed and explained in Appendix C “cmtjava Error Messages”.
- The error messages of the cmtjava2html tool are listed and explained in Appendix D “cmtjava2html Error Messages”.

1.2. About This Version of CMTJava

This CMTJava, Complexity Measures Tool for Java, version 4.0, follows the previous CMTJava v3.0. Compared to the previous 3.0 version this 4.0 version has for example:

- JSR-379 Java Language Specification, Version 9, September 2017, was in effect when developing this CMTJava upgrade, and it is the level of Java code that this tool version is supposed to be able to measure.
- JSON report format was added, use `-j/-jf` similar as XML report.
- Added Java 8 operators `'::'` and `'->'`
- Support for non-trivial enums

```
enum JavaType {
  B(‘‘Byte’’, new Byte((byte) -1), new Byte((byte) 0)),
}
```

- Support for unicode escaped characters in keywords `\u0070ublic class pp {}`
- Support for non-Ascii unicode characters in identifiers `void X\u2620\u2620ethod() {};`
- Support for unicode line-breaks

```
class foo {
\u000d
}
```

- Various fixes for additional, but unneeded, `;` semicolons and `,` commas.
- Fix for anotations in unaware places:

```
String [] @MRtnB [] array2Second() { return null; }
```

- Recognition for `module` keyword, but only in files named `module-info.java`

Read full version history from the `version.txt` file at the tool installation directory, normally pointed by `CMTJAVAHOME` environment variable.

1.3. About The Previous Versions of CMTJava

CMTJava, Complexity Measures Tool for Java, Version 1.0, was the first publicly released version of the tool (December 2001). It was worked up as a Java adaptation of the tool CMT++, Complexity Measures Tool for C/C++, Version 3.2.

Intermediate versions 1.0.1 (January 2002) and 1.0.2 (February 2002) were released and contained some small bug corrections.

In version 1.2 (April 2002) the tool was taught to measure methods of nested classes and interfaces.

In version 1.3 (August 2002) smaller enhancements were made and on the Windows platform the CMTJava GUI was introduced. Later, on Windows platform, two upgrades were released, where the GUI component was improved.

In version 1.4 (January 2004) `cmtjava2html` was introduced. Also some other improvements to the general cmtjava tool and in the CMTJava GUI (Windows platform) were done.

In version 2.0 (September 2005) support for Java 5 was added. Maintainability Index (MI) was added. Separate alarm limits for second level classes and interfaces was added. Cmtjava2xml tool was added.

In version 2.1 (September 2006) there was many bug fixes and enhancements in the basic `cmtjava` tool and in the `cmtjava2html` components.

In version 2.2 (December 2008) the "long report" (`-l` and `-lf` options) was changed to XML format. Also other enhancements and bug fixes.

In version 3.0 (September 2012) the "long report" (`-l` and `-lf` options) was changed to XML format. Also other enhancements and bug fixes.

2. Introducing CMTJava

2.1. About CMTJava and Complexity Metrics

CMTJava – Complexity Measures Tool for Java, is a tool for analyzing the static complexity and size properties of code written in Java. Code complexity is known to correlate with the defect rate and robustness of the application program as illustrated in Figure 2.1.

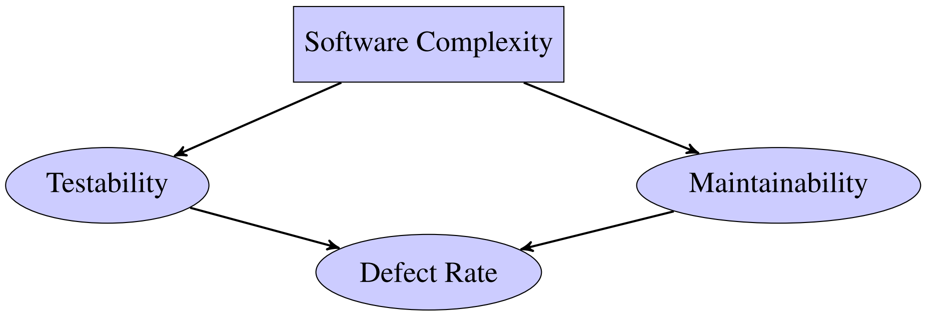

Figure 2.1.: How software complexity affects quality attributes and testing.

Figure 2.1 emphasizes firstly that complex code is difficult to test. And when it is difficult to test, probably more errors remain unrevealed in the final program. Secondly, complex code will be more error-prone in itself and affect the defect rate of the final program. And thirdly, complex code is difficult to maintain. And being so, again, it is likely that more errors find their way to the final program.

There are also cost aspects here, because the testing and maintenance are major sources of the costs in software projects. The costs of bad quality and erroneous programs can be very high, sometimes crucial to a company. Some of these costs can be attributed to unnecessarily complex code.

Now, the question is: Do we have any means to locate the complex code so that we could avoid these risks.

CMTJava is a tool, which can be used for measuring the complexity of Java code. The measures include McCabe's cyclomatic number, various lines-of-code metrics, various Halstead's metrics and Maintainability Index (MI).

These measures can be used in assessing the quality of Java code files. Based on the static properties of the program code, CMTJava gives estimates how error prone the program source code is due to its complexity, how long it will take to understand the code, what is the logical volume of the code, etc. The project team usually has not time to inspect all the code produced by the project. CMTJava can assist in locating the modules which are most likely to cause problems in the future.

The oldest way to estimate the complexity of a program is to count the number of source lines. However, this measure depends on the formatting of the code, on the programming language, on the programming style, and is insensitive to the actual logic of the program.

Several more specific and accurate measures have been developed, for example the following, all reported by CMTJava:

- the number of actual program lines (*LOCpro*), where pure comments and blank lines are ignored
- the cyclomatic number ($v(G)$), which measures the number of conditional branches in the flow of control
- the program volume (V), which is a measure of the information contents of a program
- estimate of the number of programming errors (B)
- estimate of the time needed to implement or understand the program code (T)

No measure as such is a magic number that can distinguish good programs from bad ones. But a set of different characteristic values can be used for filtering out potential candidates for further inspection.

2.2. Measures Calculated by CMTJava

CMTJava reads in a set of Java files and calculates the following software measures of them:

- McCabe's cyclomatic number
- Lines of code metrics
- Halstead metrics
- Maintainability Index
- Some other measures: number of semicolons, number of Java comment blocks (`/** ... */`), maximum nesting depth of `{...}`s, number of method parameters.

The McCabe cyclomatic number is one entity:

$v(G)$ The cyclomatic number. This measure estimates the control flow complexity of the code. This measure can be calculated in four flavors as `basic`, `extended` (default), `basic_modified`, `extended_modified`. Read more from Appendix B "How the Measures Are Calculated".

The lines-of-code measures are the following:

LOCphy The number of physical lines.

LOCpro	The number of lines with program code. (These lines may also contain comments.)
LOCcom	The number of lines with comments. (These lines may also contain program code.)
LOCbl	The number of blank lines.

The following Halstead measures are calculated:

N1	Number of operators.
N2	Number of operands.
N	Program length ($N1 + N2$).
n1	Number of unique operators.
n2	Number of unique operands.
n	Vocabulary size or number of unique operators and unique operands ($n1 + n2$).

And the following derivative measures, which are determined with certain formulae of the previous ones. Read more from Appendix B "How the Measures Are Calculated".

V	Program volume or information contents of the program.
B	Number of delivered bugs. This quantity is an estimate of the number of bugs in the program.
D	Difficulty level, error proneness.
E	Effort to implement.
L	Program level. This quantity represents the abstraction level of the program.
T	Implementation time (or time to understand).

The lines-of-code, McCabe and Halstead measures are further calculated with certain formulae to an MI (Maintainability Index) measure.

MI	Maintainability Index.
MIwoc	Maintainability Index without comments.
MIcw	Maintainability Index comment weight.

2.3. CMTJava Report Forms

CMTJava can show the measures in various forms, as follows:

Short Report

Short report is the default report form. It is a text file, meant to be reviewed by a text editor at a screen or printed on paper. It contains the most important measures in a compact tabular form (one line per measured class/interface/method): $v(G)$, *LOCphy*, *LOCpro*, commenting percent, V, B and *MI* (or *MIwoc*). A warning mark '-' is appended to a measured value, which is outside of its alarm limits.

Long Report (in XML)

Long report form is taken by applying `-l` or `-lf` option in the tool run. As of version 2.2 the report format is XML. Long report contains all the measures that cmtjava can produce, also the bottom-line summaries and the used alarm limits. When `-lf` option (long with frequencies) has been used, the report contains also a list of the operands and operators and their frequencies (based on which the Halstead measures are calculated).

Long Report (in JSON)

Long report form is taken by applying `-j` or `-jf` option in the tool run. As of version 4.0 the report format is JSON. Long report contains all the measures that cmtjava can produce, also the bottom-line summaries and the used alarm limits. When `-jf` option (json with frequencies) has been used, the report contains also a list of the operands and operators and their frequencies (based on which the Halstead measures are calculated).

Excel Report

Excel report form is taken by applying `-x` option in the tool run. Excel report contains all the measures, arranged on lines (one line per measured item), fields separated with a tab character. Excel report can be inputted to Excel (or to similar spreadsheet program) for further processing.

HTML Report

HTML report form is derived from the short report form with the `cmtjava2html` utility. The resultant HTML report can be viewed with normal web browsers. When the HTML browsing starts, a summary view, which contains measures of the two top level classes and interfaces is shown. The item names are links to a detailed view, which shows the HTML'ized form of the short report. In the detailed view the actual java source code files (the lines of the measured items) can be loaded/unloaded to the window.

In the HTML report the measures that are outside of their warning levels are marked in red.

2.4. CMTJava Reporting Levels

In one CMTJava run there can be one or more Java source files as input. You can easily arrange (see chapter 5 "Using CMTJava") all Java source files of the whole project to be measured in one CMTJava run.

CMTJava calculates measures on the following summary levels:

1. Measures over all input files (system level)
2. Measures over each single input file
3. Measures over each file-level class or interface
4. Measures over each class or interface that is immediately within the file-level class or interface
5. Measures over each method or yet inner class or interface, which are within the two top level class or interface

If there are classes, interfaces or methods that are on a still deeper nesting level, their measures are not calculated separately. Instead, their measures are reported along with the closest upper level, which is reported separately.

As of CMTJava v3.0 the tool started to recognize new type of `enum`'s that contained executable code, e.g. `enum E {E1, E2, ... ; void foo(){...}}`. They are reported separately, similarly like a *class*. Old type of *enum*'s, e.g. `enum E{E1, E2, E3}`, are not reported separately.

2.4.1. Measures over all Input Files

These measures are reported in the "short tabular form" report of CMTJava. An example clarifies what information is given:

```
OVERALL SUMMARY:

Methods (on two top levels)  Classes           Interfaces
===========================  =================  ==================
 554 Total                    209 Total            2 Total
   8 Average LOCpro           205 On top level     2 On top level
   8 Average LOCcom/LOCphy %  112 Extends          1 Extends
   2 Average v(G) (extended)   53 Implements       0 Implements

Packages                     Files
===========================  ====================================
   1 Total                     98 Total
                             8063 LOCphy
                             6158 LOCpro
System                        802 LOCcom
========================     1146 LOCbl
 595 v(G) (extended)         3542 ';'
 103 MI without comments       82 Average LOCphy
  19 MI comment weight         10 Average LOCcom/LOCphy %
 122 MI                       150 Java comment blocks
```

Some explanations of the above:

- Methods are recognized from two top level classes and interfaces only, not of any yet inner classes and interfaces.

- All classes and interfaces are recognized, let them be on top level or at any nesting level.
- In the “Packages” section it is reported how many package clauses with different package name were recognized.

2.4.2. Measures over a Single File

An example clarifies what information is given:

```
File   : F:\cmtjwork\v10\test\CoreJava\corejava\Day.java
Package: corejava
... snip, reporting of the file internals ...

Summary: v(G): 15 LOCphy: 254 LOCbl: 37 LOCpro: 108 LOCcom: 111 ';': 61
```

The above is taken from “short tabular form” report. Long form report has the same information but in XML form.

2.4.3. Measures over a Single Class or Interface

In the example short tabular report below, one full file is shown. The file contains three classes and one interface at top level. The class Employee contains two nested classes.

The line number is shown where the class/interface begins and ends. The class/interface is summarized at the "Summary:" line. Should there be yet inner classes, they would be reported on one line like methods at the second level, something like `yetInnerClass{} the_measures ....`

```
File   : F:\cmtjwork\v10\test\CoreJava\v1ch6\PropertyTest\PropertyTest.java
Package:

Line Measured item                  v(G) LOCphy LOCpro  c%    V     B    MI
=============================================================================
   8 class PropertyTest
   9   main()                          1     23     20  -   675  0.11    86
  32 Summary: PropertyTest             1     25     23  -   705  0.12    85

  34 interface Property
  35   get()                           1      1      0-      12  0.00   158
  36   set()                           1      1      1       20  0.00   155
  37   name()                          1      1      1       12  0.00   158
  38 Summary: Property                 1      5      5       72- 0.01   146

  40 class PropertyEditor
  41   PropertyEditor()                1      3      2       37  0.01   134
  45   editProperties()                6     16     16  -   521  0.14    92
  63 Summary: PropertyEditor           6     24     22  -   642  0.18   100

  65 class Employee
  66   Employee()                      1      5      4       96  0.02   121
  71   print()                         1      4      4       86  0.01   125
  75   raiseSalary()                   1      3      3       57  0.01   132
  78   hireYear()                      1      3      3       37  0.01   134
```

```
 82   class SalaryProperty
 83     name()                        1      3      2       32  0.01   135
 86     get()                         1      3      3       33  0.01   135
 90     set()                         3      8      8      145  0.03   137
 99   Summary: SalaryProperty         3     18     17  -   328  0.07   135
101   getSalaryProperty()             1      3      3       32  0.01   135
105   class SeniorityProperty
106     name()                        1      3      2       32  0.01   135
109     get()                         1      5      5      121  0.03   120
114     set()                         1      2      2       20  0.00   144
116   Summary: SeniorityProperty      1     12     12  -   250  0.06   147
118   getSeniorityProperty()          1      3      3       32  0.01   135
125 Summary: Employee                 3     61     55  -  1328  0.29   119
=============================================================================
Summary: v(G): 8   LOCphy: 125  LOCbl: 15   LOCpro: 106  LOCcom: 6   ';': 46
```

In a “long report” and in an “excel report” there is a bit more information shown.

2.4.4. Measures over a Single Method and Inner Class/Interface

The previous example clarifies this as well. The meaning of the columns are:

`Line:` Tells the line number where the measured item begins (first `LOCpro` line) in the source file. Configuration parameter `NOTICE_LEADING_COMMENTS` determines if the possible leading comment block is associated to the measured item, is calculated to its lines, and where the item starts. On class/interface summary this column tells the item’s end line.

`Measured item`: What class/interface/method we have.

`v(G)`: McCabe Cyclomatic number (control flow complexity)

`LOCphy`: How big the item is in terms of source lines.

`LOCpro`: How many program lines the item has.

`c%`: Warning of a too low or too high commenting percent (`LOCcom/LOCphy`). The column value is either empty (no warning) or ’`-`’ (warning). The rules for determining the warning are: First, if the item is a small one (limit is defined in the configuration parameter `NO_COMMENT_WARNINGS_BELOW` in terms of `LOCpro`), the item is “excused”, i.e. is never alarmed for its commenting. But when the item is “big enough” a warning is given if the commenting ratio is outside of configured acceptable limits or if the item does not absolutely have at least certain many comment lines. The used limit values are shown at the end of the “short tabular form” report.

`V`: Halstead volume. Characterizes how big the item is. This measure is calculated from the amount of operands and operators; the effect of comments, code layout on lines and the length of the used identifiers has been filtered away.

B: Hal stead estimated number of bugs. How may bugs the code might contain as estimated from it's operator/operand complexity.

MI (or MIwoc): Maintainability Index (or Maintainability Index without comments). A single-value quantification (or index) or the relative maintainability of the item, calculated with certain formulae of the item's McCabe and Halstead measures (and of the commenting ratio).

Also here, with a "long report" and with an "excel report" some more information of a single method or inner class/interface is shown, the following in a "long report": v(G), LOCphy, LOCpro, LOCbl, LOCcom, N, N1, N2, n, n1, n2, V, B, D, E, L, T, MIwoc, MIcw, MI.

2.5. CMTJava Warnings

A warning here means that the measured value is outside of the limits that are specified in CMTJava configuration file and an *alarm* of it is displayed. In short report character '-' after the measure is used to indicate a warning.

The previous example showed warning only of commenting ratio (c\%) of the items. Potentially there could have been warning also on measures v(G), LOCpro, V, B and MI (or MIwoc).

The used alarm limits are shown at the end of the "short report" as follows:

```
============================================================================
v(G)          765       8   1       1-100            1-100         1-10
LOCpro        765      15   1       4-400            4-400         1-40
Comment %     765     317  41      20-60            20-60         20-60
V             765      37   4     100-8000         100-8000        4-1000
B             765       1   1       0-2              0-2           n/a
MI            765      33   4      80-              80-           80-
==========================================================================
Total        4590     243   5
```

2.6. CMTJava Tool Components

cmtjava The "basic CMTJava tool", which reads the input source files and produces a textual report of them.

cmtjava2html A Perl utility for converting a textual CMTJava report to HTML form, containing the input Java files as HTML'ized copies or as links to them.

cmtjavaui (on Windows only) A CMTJava GUI for using the CMTJava tool components graphically.

3. Installing CMTJava

On Windows the installation is performed by a *setup* program (InstallShield). On Unix platforms the installation is performed by a *makefile*, which you can edit to meet your needs and then execute for copying the tool files to their correct places.

One of the delivered files is `INSTALL.TXT` or `README.TXT`. You should view it before starting the actual installation. It gives you the necessary instructions for performing the actual installation on your platform.

The CMTJava requirements for installation and use are rather modest. Your normal Java development environment is quite sufficient. Depending on the platform, 2–4 MB free disk space is enough.

You need not have a Java compiler available in the environment where you use CMTJava.

4. Configuring CMTJava

CMTJava's behavior can be tuned by modifying the configuration parameters in the configuration file cmtjava.ini with any text editor. By tuning here it is meant changing the CMTJava alarm limit parameters and other settings.

The configuration file contains also certain license control settings. Sometimes these settings need to be modified, for example when installing an evaluation copy license or when advising how the tool finds connection to the Flexlm license manager server.

The definition of a configuration parameter has the following syntax:

```
        PARAMETER-NAME=PARAMETER-VALUE
```

A line whose first non-whitespace character is '#' is comment. Empty lines can be used.

Parameter names are case-sensitive. Some of the parameters are used for the software license identification, and they must not be modified by the user.

You can use environment variables in a configuration file. A construct `$(ENV_VAR_NAME )`' is replaced with the value of the corresponding environment variable (with empty, if the environment variable is not known).

CMTJava searches and reads configuration files from a number of places. See section 5.2 "Starting CMTJava from the Command Line" for a description of the command line parameter `-c` (configuration) and for the searched locations. A configuration parameter definition read later (from the same file or from another file) overrides any previously read definition.

Additionally, with `-C` command line parameter, a configuration file setting may be overridden from the command line.

The configuration parameters (except the license control parameters) can be edited as felt appropriate.

The configuration parameters are described below.

4.1. Measure Alarm Limit Parameters

Most configuration parameters are used to define the acceptable limits of the complexity measures.

There are three sets of these alarm limit type of configuration parameters. Parameter name of type "`..._CLASS`" applies on top level classes (and enums) and interfaces. Parameter name of type "`..._CLASS2`" applies on second level classes (and enums) and interfaces. Parameter name of type "`..._METHOD`" applies on all methods and on yet inner classes and interfaces.

If a calculated measure is outside the specified limits, CMTJava marks it with a minus sign, '`-`', in the report.

The limit parameters are listed below.

The limits are discussed further in the chapter 6.

- Specify the lowest and highest acceptable value for the McCabe's cyclomatic number ($v(G)$) on a measured item. The $v(G)$ value itself is calculated according to the rules as the configuration parameter `MCCABE_PREFERENCE` determines.

```
MCCABE_CLASS_MIN
MCCABE_CLASS2_MIN
MCCABE_METHOD_MIN
MCCABE_CLASS_MAX
MCCABE_CLASS2_MAX
MCCABE_METHOD_MAX
```

- Specify the lowest and highest acceptable number of executable lines (*LOCpro*) of a measured item.

```
LOC_CLASS_MIN
LOC_CLASS2_MIN
LOC_METHOD_MIN
LOC_CLASS_MAX
LOC_CLASS2_MAX
LOC_METHOD_MAX
```

- Specify the lowest and highest acceptable values for the ratio of comment lines and total source code lines for the measured item. The value is a percentage number ($100 \cdot LOCcom/LOCphy$), no decimals.

```
COMMENT_RATIO_CLASS_MIN
COMMENT_RATIO_CLASS2_MIN
COMMENT_RATIO_METHOD_MIN
COMMENT_RATIO_CLASS_MAX
COMMENT_RATIO_CLASS2_MAX
COMMENT_RATIO_METHOD_MAX
```

- Specify whether the possible comment block should be counted to the lines of the measured item that follows. Value 1 means yes, value `0` means no. This algorithm is a bit heuristic. The preceding comment block is recognized and associated to the item that follows it, if there are no other code lines between the comment block and the beginning of the item. After the comment block there can be zero or more empty lines. If there is an intermediate empty line (that is not enclosed in `/*... */` commenting) between the comment block, it breaks the comment block counting and association to the following item. There can be both `/*...*/` comments and `//` comments in the comment block

```
NOTICE_LEADING_COMMENTS
```

- If the measured item is smaller than the given argument (in *LOCpro*), no absolute comment line amount or comment line ratio warnings are given of the item.

```
NO_COMMENT_WARNINGS_BELOW
```

- Specify the absolute minimum number of comment lines (*LOCcom*) that a method or third level class/interface should have.

```
COMMENT_METHOD_MIN
```

- Specify the lowest and highest acceptable volume (V) of a measured item.

```
V_CLASS_MIN
V_CLASS2_MIN
V_METHOD_MIN
V_CLASS_MAX
V_CLASS2_MAX
V_METHOD_MAX
```

- Specify the lowest and highest acceptable number of errors of a measured item. “Error” means here the value of the complexity measure estimating the error-proneness (*B*) and where the `B_CORRECTION_FACTOR` value has been applied. Note that for methods there are no *B* alarm limits.

```
B_CLASS_MIN
B_CLASS2_MIN
B_CLASS_MAX
B_CLASS2_MAX
```

- Argument is a positive decimal number, default 1.0. The *B* value (estimated number of bugs) is first calculated as it is defined by Halstead. Then, before displaying the *B* value to the reports and comparing it against the *B*-limits, it is multiplied with the given correction factor. Some authors have mentioned correction factor value 1.95, which you could also consider, if you decide to take this policy.

```
B_CORRECTION_FACTOR
```

- Specify the lowest acceptable *MI* (or *MIwoc*) value for a measured item. The limit applies either to the *MI* value or to the *MIwoc* value depending on the `MI_PREFERENCE` selection.

```
MI_CLASS_MIN
MI_CLASS2_MIN
MI_METHOD_MIN
```

- Specify if *MI* or *MIwoc* is displayed in short report and do the limit values apply on the calculated *MI* or *MIwoc* values. The argument must be either `MI` or `MIwoc`.

```
MI_PREFERENCE
```

- Determines the “flavor” how the $v(G)$ is calculated. The alternatives are: `basic`, `extended` , `basic_modified`, `extended_modified`. Read more from Appendix B “How the Measures Are Calculated”.

```
MCCABE_PREFERENCE
```

4.2. Excel Field Separator

With the `-x` option CMTJava produces its output to a text file in a form, which is suitable for Excel (or a similar spreadsheet utility) input file. With setting

```
EXCEL_FIELD_SEPARATOR
```

Specify the ascii value of the character, which will be used as a field separator in the file. For example, the value `9` means tab character ('`\t`') and `59` means a semicolon ('`;`').

4.3. Software License Parameters

The configuration parameters `TOOL`, `USER`, `COMPUTER`, `LICENCE`, `EXPIRATION`, `NOTE1`, `NOTE2` , `NOTE3`, `NOTE4`, `NOTE5`, `TARGET_CHECK`, and `CONTROL` are used for identifying the CMTJava software license. Do not modify their settings. If they are modified in an unauthorized way, CMTJava will not work any more.

4.4. Hardware Control Key Port

This parameter is needed only in environments (PC), where a hardware control key, which is plugged into a parallel port, can be used to control the license. The port number is the number of the user selectable LPT port where the control key is attached to. For example, the following definition is correct.

```
KEYPORT=1
```

As of v2.2 the tool is no more delivered so licensed that it would require a hardware control key (dongle). This was available only at Windows.

4.5. Link to Floating License Manager

In some environments the license may be controlled by FLEXlm license manager. Parameter `FLEXLM_LICENSE_FILE` specifies the path and name of the `testwell.lic` license file, or port and host of the license manager daemon. This parameter is not required if FLEXlm licensing is not used. Examples

```
FLEXLM_LICENSE_FILE=@FlxServer
FLEXLM_LICENSE_FILE=27000@FlxServer
FLEXLM_LICENSE_FILE=/usr/local/flexlm/licenses/testwell.lic
FLEXLM_LICENSE_FILE=$(CMTJAVAHOME)\testwell.lic
```

5. Using CMTJava

5.1. Overall Architecture

Simply, CMTJava is a tool, which analyzes Java source files for their size, complexity and maintainability properties (see Figure 5.1). The input files normally comprise one program or project, but in principle they can be totally unrelated to each other. The textual output file format depends on the used command-line options.

Some output file forms are meant to be directly usable/viewable by the user. One output file form can be further converted (by cmtjava2html tool) to HTML form. One textual file form can be directly inputted to Excel (or similar spreadsheet program).

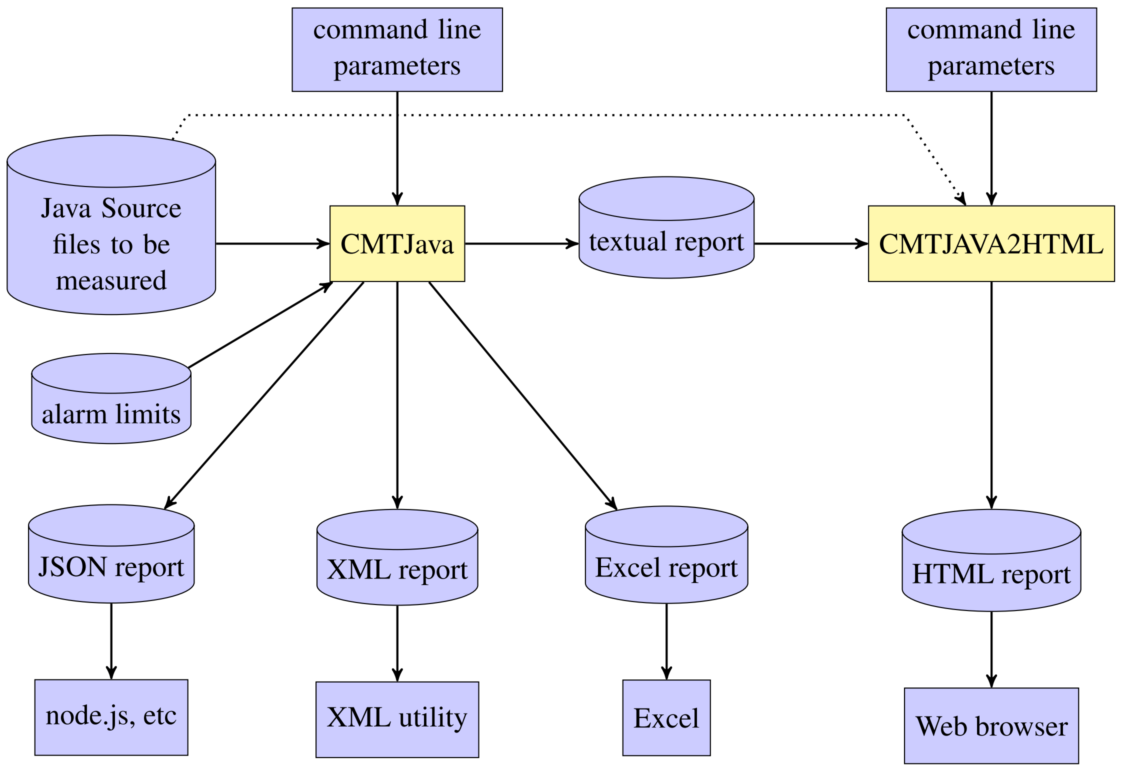

Figure 5.1.: The I/O connections of the CMTJava tool.

5.2. Starting CMTJava from the Command Line

The basic CMTJava tool is started from the command line in one of the following ways:
For getting a short or a somewhat longer on-line help:

```
cmtjava -h | -H
```

For producing a short form report:

```
cmtjava [general-opts] [-s] [-w] [-f filenames] [-o outfile] [sourcefile...]
```

For producing an Excel input file form report:

```
cmtjava [general-opts] -x [-nxh] [-s] [-w] [-f filenames] [-o outfile] [sourcefile...]
```

For producing a long report (XML form):

```
cmtjava [general-opts] -l[f] [-s] [-f filenames] [-o outfile] [sourcefile...]
```

For producing a long report (JSON form):

```
cmtjava [general-opts] -j[f] [-s] [-f filenames] [-o outfile] [sourcefile...]
```

Where the general-opts are:

```
[-c conffiles] [-C confparam=value]... [-v]
```

The above four alternatives to invoke CMTJava specify also the allowed combinations of the options.
The meanings of the command-line options and arguments are:

`-h` ("small help") Displays a synopsis of the command line options. If the `-h` option is given, the command line must not contain other arguments.

`-H` ("big help") Displays a few screens of general help about CMTJava. If the `-H` option is given, the command line must not contain other arguments.

`-c conffiles` ("additional configuration files") Specifies one or more configuration files to be read by CMTJava after the default locations for configuration files have been searched. If more than one file is given, they must be separated by a semicolon. There must be one space after the `-c` option but, if multiple configuration files are specified, no spaces around the '`;`'s.

The configuration files are searched from the following places in order (a later definition overrides a previous one):

1. File `/usr/local/lib/cmtjava/cmtjava.ini` (Unix only).
2. File `\$HOME/lib/cmtjava/cmtjava.ini` (Unix only)
3. File `cmtjava.ini` in the directory specified by the `CMTJAVAHOME` environment variable.
4. file `.cmtjava.ini` in the user's home directory (Unix only).

5. file specified by the environment variable `CMTJAVAINIT`. Multiple files can be specified, separated by a semicolon.
6. File cmtjava.ini in the current directory (file `.cmtjava.ini` in Unix).
7. File(s) explicitly specified by the `-c` option.

`-C confparam=value` Override configuration parameter as it was read from the configuration file(s). There can be many `-C` options.

`-v` (“verbose”) Show on the screen what configuration files CMTJava tried to find and loaded if the file was found.

`-s` (“summary”) Top level class/interface summary only. In the absence of this option the output file contains measures both of the top level classes/interfaces and of their internal items. (If the `-x` option is also present, the behavior is slightly different, see below)

`-w` (“warnings only”) Only lines that have a warning flag '`-`' are written to the output file. Usable when producing the short report or Excel data file.

`-x` (“excel output”) The output file produced is of the Excel input data file format. Normally, the `-s` option is not present together with this `-x` option. In such a case the resultant Excel data file contains only measures of the methods (+ possible classes/interfaces that are on the third nesting level). If both the `-x` and `-s` options are present, the Excel data file contains only the summary level data of top level classes/interfaces.

`-nxh` (“no excel header”) When producing the Excel data file output, no column headers are written to the first line of the file, only the actual data lines.

`-l` (“long”) CMTJava produces the complexity measure report in long format in XML where all measures are included.

`-lf` (“long with frequencies”) CMTJava produces the complexity measure report in long format as with the `-l` option but includes also operator and operand frequencies in the report.

`-j` (“json long”) CMTJava produces the complexity measure report in long format in JSON where all measures are included.

`-jf` (“json long with frequencies”) CMTJava produces the complexity measure report in json long format as with the `-j` option but includes also operator and operand frequencies in the report.

`-f filenames` (“file names”) Specifies a text file that has the names of the files to be measured, one file name on a line. If the `-f` option is present, there no more can be explicit source file names on the command line.

`-o outputfile` (“output file”) Specifies the output file. If the `-o` option is present, it must be followed by a filename separated by a space. The complexity measure report is then written to that file (and silently overwriting the possible previous file with that name). In the absence of this option the report is written to *stdout*.

`sourcefile...` A list of the file names (separated by a space) to be measured. Wildcards can be used. Character '`-`' means the file *stdin*, i.e. the file to be measured is read from *stdin*. If no source files are given, CMTJava prompts for the file names interactively.

For example, the command

```
cmtjava *.java
```

tells CMTJava to analyze all files in the working directory that have the extension `.java`. The report is written to *stdout*. Whereas the command

```
cmtjava -c prjlims.ini -lf -o report.xml *.java reuse*.java
```

tells CMTJava to

- analyze all `.java` source files in the working directory and the files whose names start with `f` and whose extension is `.java` in the subdirectory reuse.
- use the additional (actually overriding) configuration settings from the file `prjlims.ini`.
- produce the report to the file `report.xml` (XML format).
- report all possible measurements (long listing, operand frequencies included).

One more example, use of `-C` option, where certain configuration parameters are overridden (enforced) from command line:

```
cmtjava -w -C NO_COMMENT_WARNINGS_BELOW=1
           -C COMMENT_METHOD_MIN=5 -o report.txt file?.java
```

5.3. Using CMTJava Interactively

If no source files are given on the command line, CMTJava prompts them interactively from *stdin*. For example

```
C:\TESTDIR> cmtjava -o report.txt
******************************************************************************
*          CMTJava, Complexity Measures Tool for Java, Version 4.0          *
*                                                                            *
*                    Copyright (c) 2001-2012 Testwell Oy                     *
*               Copyright (c) 2018 Verifysoft Technology GmbH                *
******************************************************************************

Invoke the tool with option -h for help about CMTJava.
Type '?' in a prompt for context sensitive help.
Type '!' in a prompt for visiting operating system.
```

```
Names of the source code files?
SOURCE ( 1) => file1.java
SOURCE ( 2) => file2.java
SOURCE ( 3) =>

Measuring file 1 2 Done

C:\TESTDIR>
```

The `SOURCE ( n)=> prompt` is repeated until your answer with an empty line (or EOF is met in the reading). Here two files `file1.java` and `file2.java` are measured and the results are written to the file `report.txt`. The result file is of short form (neither `-x` nor `-l/-lf/-j/-jf` options were present).

You could have got the same behavior directly from the command line with the command

```
C:\TESTDIR> cmtjava -o report.txt file1.java file2.java
```

or as follows

```
C:\TESTDIR> cmtjava file1.java file2.java > report.txt
```

or, if wildcard notation `file?.java` matches only to these two files, as follows

```
C:\TESTDIR> cmtjava -o report.txt file?.java
```

In the `SOURCE (i)=>` prompt possible answers are also:

`!` A new operating system shell is started and in it any commands can be executed. Finally, the shell is ended with '`exit`' command and the `SOURCE (i)=>` prompt is repeated.

`!command` The given operating system command, for example '`dir *.java`', is executed and the `SOURCE (i)=>` prompt is repeated.

`#command` Considered comment, the `SOURCE(i)=>` prompt is just repeated.

5.4. Piping Source File Names to CMTJava

This style of using CMTJava is actually a variant of using CMTJava interactively. The difference is that the source file names are read from a file, which is piped to *stdin*, instead of typing the file names interactively from the terminal. An example:

```
C:\TESTDIR> cmtjava -o report.txt < files.lst
```

which is effectively the same as

```
C:\TESTDIR> cmtjava -f files.lst -o report.txt
```

You have prepared a text file `files.lst`, which contains the names of the files to be measured, one file name at a line. CMTJava believes to be running in the interactive mode and reads the answers to the `SOURCE ( n)=>` prompts from the piped file. `EOF` (or when an empty line is met) ends the source file name reading.

You can comment out some files by prefixing the file names with '\#' character.

Another example:

```
C:\TESTDIR> dir *.java /s /b | cmtjava -o report.txt
```

Here with the operating system command a bare file name listing of `*.java` files is produced from the current directory including its subdirectories. The file name list is piped to CMTJava, which produces the `report.txt` file.

Note that the following

```
C:\TESTDIR> dir *.java /s /b | cmtjava > report.txt
```

will **not** work smoothly, because the interactive mode tool header box and source file prompting texts get also copied to the output file.

5.5. Reading the Actual Source File from *stdin*

Recall that '`-`' as a source file name means *stdin*. This being so, the following two commands are functionally equivalent:

```
C:\SOURCEDIR> type file1.java | cmtjava -o report.txt -
C:\SOURCEDIR> cmtjava -o report.txt file1.java
```

Well, there is a slight difference: in the first form CMTJava thinks to be reading a file named "`-`" (CMTJava's escape notation for *stdin*) while in the latter form CMTJava correctly knows that it reads the file `file1.java`.

5.6. Example

Here is a complete example of measuring the complexity of two Java files. First *Stack.java*:

```
/**
* Class Stack provides a general purpose object stack.
*/

public class Stack
{

    private int myheight;
    private int size;
    private Object value[];

    private static final int SIZE_STEP = 10;

    /**
    * Constructor
    */
    public Stack()
    {
        myheight = 0;
        size = 0;
        value = null;
    }


    /**
    * Clears the stack
    */
    public void clear()
    {
        myheight = 0;
    }


    /**
    * Returns the height of the stack
    *
    * @return height of stack
    */
    public int getHeight()
    {
        return myheight;
    }

    /**
    * Pushes an object into stack.
    *
    * @param item the object to be pushed
    */
    public void push(
        Object item)
    {
        if (myheight >= size)
        {
            size += SIZE_STEP;
            Object new_value[] = new Object[size];
            for (int i = 0; i < myheight; i++)
                new_value[i] = value[i];
            value = new_value;
        }
        value[myheight++] = item;
    }

    /**
    * Pops the topmost item of the stack and returns it
    *
    * @return topmost object in the stack
    */
    public Object pop()
    {
        Object item = null;
        if (myheight > 0)
        {
            item = value[--myheight];
            value[myheight] = null;
        }

        return item;
    }

    /**
    * Returns the topmost item in the stack
    *
    * @return topmost object in the stack
    */
    public Object top()
    {
        Object item = null;

        if (myheight > 0)
            item = value[myheight - 1];

        return item;
    }

    /**
    * Returns the specified item in the stack
    *
    * @param index the index of the object to be returned
    * @return specified object in the stack
    */
    public Object getElement(int index) {
        Object item = null;

        if (index >= 0 && index < myheight)
            item = value[index];

        return item;
    }

}
```

Then *StackTest.java*:

```
public class StackTest
{
    public static void main(String args[])
    {
        Stack stack = new Stack();
        stack.push(new Integer(10));
        stack.push(new Integer(20));
        System.out.println(stack.top());
        for (int i = 0; i < stack.getHeight(); i++)
        {
            System.out.println(stack.getElement(i));
        }
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

We now measure the complexity of these files:

```
C:\TESTDIR>cmtjava -o report.txt Stack.java StackTest.java
**********************************************************************
*      CMTJava, Complexity Measures Tool for Java, Version 3.0       *
*                                                                    *
*                Copyright (c) 2001-2012 Testwell Oy                 *
**********************************************************************

Measuring file 1 2 Done

C:\TESTDIR>
```

The resulting report file `report.txt` is of a “short tabular form”:

```
**********************************************************************
*       CMTJava, Complexity Measures Tool for Java, Version 3.0      *
*                                                                    *
*                      COMPLEXITY MEASURES REPORT                    *
*                                                                    *
*                 Copyright (c) 2001-2012 Testwell Oy                *
**********************************************************************

This report was produced at Mon Sep 10 06:40:50 2012
Options: -o report.txt


File   : Stack.java
Package:

Line Measured item              v(G) LOCphy LOCpro  c%   V     B    MI
======================================================================
   1 class Stack
  14   Stack()                     1      9      6      55  0.01   153
  25   clear()                     1      7      4      29  0.01   164
  34   getHeight()                 1      9      4      24  0.01   164
  44   push()                      3     18     13     260  0.08   131
  63   pop()                       2     17     10     127  0.04   137
  81   top()                       2     14      7     104  0.03   144
  96   getElement()                3     14      6     125  0.04   145
 111 Summary: Stack                7    111     57    1086  0.45   138
======================================================================
Summary: v(G): 7  LOCphy: 111 LOCbl: 19 LOCpro: 57 LOCcom: 35 ’;’: 26


File   : StackTest.java
Package:

Line Measured item             v(G) LOCphy LOCpro  c%   V     B    MI
```

```
======================================================================
   1 class StackTest
   3   main()                     2     13     13  -  475  0.11    97
  16 Summary: StackTest           2     16     16  -  504  0.12    93
======================================================================
Summary: v(G): 2  LOCphy: 16  LOCbl: 0  LOCpro: 16  LOCcom: 0 ';': 9


OVERALL SUMMARY:

Methods (on two top levels)     Classes               Interfaces
==============================  ===================== ================
     8 Total                         2 Total             0 Total
     7 Average LOCpro                2 On top level      0 On top level
    31 Average LOCcom/LOCphy %       0 Extends           0 Extends
     1 Average v(G) (extended)       0 Implements        0 Implements

Packages                        Files
==============================  ======================================
     0 Total                      2 Total
                                127 LOCphy
                                 73 LOCpro
System                           35 LOCcom
==============================   19 LOCbl
     8 v(G) (extended)           35 ';'
   103 MI without comments       63 Average LOCphy
    38 MI comment weight         28 Average LOCcom/LOCphy %
   142 MI                         8 Java comment blocks

                                     Limits           Limits          Limits
Measure Measured Alarmed  % 1st level class 2nd level class   method
======================================================================
v(G)          10       0  0      1-100           1-100          1-10
LOCpro        10       0  0      4-400           4-400          1-40
Comment %     10       2 20     20-60           20-60          20-60
V             10       0  0    100-8000       100-8000          4-1000
B             10       0  0      0-2             0-2            n/a
MI            10       0  0     80-             80-            80-
======================================================================
Total         60       2  3
```

From the report we can conclude that CMTJava does not complain much about the code, only some commenting percents were outside the used limits. At the time of this measurement run there was the `NOTICE_LEADING_COMMENTS=1` setting in effect, i.e. the method's preceding comment lines were counted to the lines of the method that followed.

Long report (in XML form) is produced as follows:

```
C:\TESTDIR> cmtjava -l -o report.xml Stack.java StackTest.java
```

And the `report.xml` file looks as follows:

```
<?xml version="1.0"?>
<cmtjava_long_report>
        <header_info>
```

```
                <cmtjava_version>4.0</cmtjava_version>
                <copyright>Copyright (c) 2001-2012 Testwell Oy, Copyright 2018 Verifysoft Technology GmbH</copyright>
                <license_notes>
                </license_notes>
                <date>Tue Mar 27 11:44:19 2018</date>
                <cmtjava_options>-l -o report.xml</cmtjava_options>
                <cwd>F:\cmtjwork\v40\doc</cwd>
                <mccabe_preference>extended</mccabe_preference>
        </header_info>
        <file name="Stack.java" stamp="1522139437">
                <package></package>
                <imports>
                </imports>
                <class name="Stack" nesting="0" start_line="1">
                        <method name="Stack" nesting="1" start_line="14">
                                <vG_b>1</vG_b>
                                <vG_e>1</vG_e>
                                <vG_b_max>1</vG_b_max>
                                <vG_e_max>1</vG_e_max>
                                <vG_b_avg>1</vG_b_avg>
                                <vG_e_avg>1</vG_e_avg>
                                <params>0</params>
                                <LOCphy>9</LOCphy>
                                <LOCpro>6</LOCpro>
                                <LOCbl>0</LOCbl>
                                <LOCcom>3</LOCcom>
                                <N>16</N>
                                <N1>9</N1>
                                <N2>7</N2>
                                <n>11</n>
                                <n1>5</n1>
                                <n2>6</n2>
                                <V>55.351</V>
                                <B>0.010</B>
                                <D>2.917</D>
                                <E>161.440</E>
                                <L>0.343</L>
                                <T>00:00:08</T>
                                <MaxND>1</MaxND>
                                <MIwoc>114</MIwoc>
                                <MIcwc>39</MIcwc>
                                <MI>153</MI>
                        </method>
                        <method name="clear" nesting="1" start_line="25">
... etc., similar information blocks of other methods until there comes.
                        <class_total>
                                <vG_b>6</vG_b>
                                <vG_e>7</vG_e>
                                <vG_b_max>3</vG_b_max>
                                <vG_e_max>3</vG_e_max>
                                <vG_b_avg>1</vG_b_avg>
                                <vG_e_avg>1</vG_e_avg>
                                <params>2</params>
                                <LOCphy>111</LOCphy>
                                <LOCpro>57</LOCpro>
                                <LOCbl>19</LOCbl>
                                <LOCcom>35</LOCcom>
                                <N>199</N>
                                <N1>107</N1>
                                <N2>92</N2>
                                <n>44</n>
                                <n1>22</n1>
                                <n2>22</n2>
                                <V>1086.427</V>
                                <B>0.452</B>
                                <D>46.000</D>
                                <E>49975.637</E>
                                <L>0.022</L>
                                <T>00:46:16</T>
                                <MaxND>2</MaxND>
                                <MIwoc>100</MIwoc>
                                <MIcwc>38</MIcwc>
                                <MI>138</MI>
                        </class_total>
                </class>
                <file_total>
                        <vG_b>6</vG_b>
                        <vG_e>7</vG_e>
                        <LOCphy>111</LOCphy>
                        <LOCbl>19</LOCbl>
                        <LOCpro>57</LOCpro>
                        <LOCcom>35</LOCcom>
                        <semicolons>26</semicolons>
                </file_total>
        </file>
        <file name="StackTest.java" stamp="1522139837">
... etc., similar information of other functions, until there comes
        <system>
                <packages>0</packages>
                <files>2</files>
                <measured_items>10</measured_items>
                <LOCphy>127</LOCphy>
                <LOCpro>73</LOCpro>
                <LOCcom>35</LOCcom>
                <LOCbl>19</LOCbl>
                <semicolons>35</semicolons>
                <java_comment_blocks>8</java_comment_blocks>
                <vG_b>7</vG_b>
                <vG_e>8</vG_e>
                <MIwoc>103</MIwoc>
                <MIcw>38</MIcw>
                <MI>142</MI>
                <methods_on_two_top_levels>
                        <total>8</total>
                        <LOCpro_avg>7</LOCpro_avg>
                        <comment_percent_avg>31</comment_percent_avg>
                        <vG_b_avg>1</vG_b_avg>
                        <vG_e_avg>1</vG_e_avg>
                </methods_on_two_top_levels>
                <classes>
                        <total>2</total>
                        <on_top_level>2</on_top_level>
                        <extends>0</extends>
                        <implements>0</implements>
                </classes>
                <interfaces>
                        <total>0</total>
                        <on_top_level>0</on_top_level>
                        <extends>0</extends>
                        <implements>0</implements>
                </interfaces>
                <alarms>
                        <vG_alarmed>0</vG_alarmed>
                        <LOCpro_alarmed>0</LOCpro_alarmed>
                        <comment_percent_alarmed>2</comment_percent_alarmed>
                        <V_alarmed>0</V_alarmed>
                        <B_alarmed>0</B_alarmed>
                        <MI_alarmed>0</MI_alarmed>
                </alarms>
                <alarm_limits>
                        <top_level_classes>
                                <vG_low>1</vG_low>
                                <vG_high>100</vG_high>
                                <LOCpro_low>4</LOCpro_low>
                                <LOCpro_high>400</LOCpro_high>
                                <comment_percent_low>20</comment_percent_low>
                                <comment_percent_high>60</comment_percent_high>
                                <V_low>100</V_low>
                                <V_high>8000</V_high>
                                <B_low>0</B_low>
                                <B_high>2</B_high>
                                <MI_low>80</MI_low>
                        </top_level_classes>
                        <second_level_classes>
                                <vG_low>1</vG_low>
                                <vG_high>100</vG_high>
                                <LOCpro_low>4</LOCpro_low>
                                <LOCpro_high>400</LOCpro_high>
                                <comment_percent_low>20</comment_percent_low>
                                <comment_percent_high>60</comment_percent_high>
                                <V_low>100</V_low>
                                <V_high>8000</V_high>
                                <B_low>0</B_low>
                                <B_high>2</B_high>
                                <MI_low>80</MI_low>
                        </second_level_classes>
                        <methods>
                                <vG_low>1</vG_low>
                                <vG_high>10</vG_high>
                                <LOCpro_low>1</LOCpro_low>
                                <LOCpro_high>40</LOCpro_high>
                                <comment_percent_low>20</comment_percent_low>
                                <comment_percent_high>60</comment_percent_high>
                                <V_low>4</V_low>
                                <V_high>1000</V_high>
                                <MI_low>80</MI_low>
                        </methods>
                </alarm_limits>
                <error_messages count="0">
                </error_messages>
        </system>
</cmtjava_long_report>
```

The XML tags should be rather self-explanatory, but some remarks of them:

- In report header `<license notes>...</license_notes>`: If the `cmtjava.ini` configuration file has such `NOTEn` lines, they are copied here. Evaluation copy license normally has such lines.
- `<date>...</date>`: is the date when the cmtjava run was done.
- `<cwd>...</cwd>`: is the current working directory, where the cmtjava run was done. It is useful in determining the source files, if they are specified with relative names.
- In `<file name="..."stamp="...">` the stamp is file's last modification time, the one which shows e.g. in '`dir`' listings. It is useful when wanting to check that the file is unchanged since it was measured.
- In various places there are two McCabe cyclomatic numbers, `vG_b` and `vG_e`. They stand for basic and extended. As you recall CMTJava can calculate this measure in four flavors: `basic`, `extended`, `basic_modified`, `extended_modified`. McCabe `basic[_modified]` cyclomatic number is shown in `vG_b` and McCabe `extended[_modified]` cyclomatic number is shown in `vG_e`. There is no distinction whether the measure is `_basic` or not. That information can be seen from the `<mccabe_preference>...</maccabe_preference>` tag.

If the report had been produced with `-lf` (long with frequencies) option instead on `-l` only, there would have been information of operand and operators per each method and class/interface. For example of the method `Stack.clear()`, it would have been:

```
             <operators>
                <token count="1">()</token>
                <token count="1">;</token>
                <token count="1">=</token>
                <token count="1">public</token>
                <token count="1">{}</token>
            </operators>
            <operands>
                <token count="1">0</token>
                <token count="1">clear</token>
                <token count="1">myheight</token>
                <token count="1">void</token>
            </operands>
```

Long report (in JSON is produced as follows:

```
C:\TESTDIR> cmtjava -j -o report.json Stack.java StackTest.java
```

And the `report.json` file looks as follows:

```
{
        "header_info": {
                "cmtjava_version": 4.0,
                "copyright": [
                        "Copyright (c) 2001-2012 Testwell Oy",
                        "Copyright (c) 2018 Verifysoft Technology GmbH"
                ],
                "license_notes": [
                        "License notice: This is a limited period evaluation copy license."
                ],
                "date": "Tue Mar 27 12:01:58 2018",
                "cmtjava_options": " -j -o report.json",
                "cwd": "/home/roland/projects/cmtjavaug/cmtjava",
                "mccabe_preference": "extended"
        },
                "file": [
        {
        "name": "Stack.java" ,
        "stamp" : 1522139437,
                "package": "" ,
                "imports" : [
                ],
                "class" : [
                {
                "type": "class" ,
                "name": "Stack" ,
                "nesting" : 0,
                "start_line" : 1,
                "method" : [
                {
                "name": "Stack" ,
                "nesting" : 1,
                "start_line" : 10,
                                "vG_b" : 1,
                                "vG_b_alarmed" : false,
                                "vG_e" : 1,
                                "vG_e_alarmed" : false,
                                "vG_b_max" : 1,
                                "vG_e_max" : 1,
                                "vG_b_avg" : 1,
                                "vG_e_avg" : 1,
                                "params" : 0,
                                "LOCphy" : 9,
                                "LOCpro" : 6,
                                "LOCpro_alarmed" : false,
                                "LOCbl" : 0,
                                "LOCcom" : 3,
                                "LOCcom_alarmed" : false,
                                "N" : 16,
                                "N1" : 9,
                                "N2" : 7,
                                "n" : 11,
                                "n1" : 5,
                                "n2" : 6,
                                "V" :55.351,
                                "V_alarmed" : false,
                                "B" :0.010,
                                "D" :2.917,
                                "E" :161.440,
                                "L" :0.343,
                                "T": "00:00:08" ,
                                "MaxND" : 1,
                                "MIwoc" : 114,
                                "MIcwc" : 39,
                                "MI" : 153,
                                "MI_alarmed" : false
                        }
                        ,

                {
                  "name": "clear" ,
... etc., similar information blocks of other methods until there comes.
                        "class_total" : {
                                "vG_b" : 6,
                                "vG_b_alarmed" : false,
                                "vG_e" : 7,
                                "vG_e_alarmed" : false,
                                "vG_b_max" : 6,
                                "vG_e_max" : 7,
                                "vG_b_avg" : 2,
                                "vG_e_avg" : 2,
                                "params" : 1,
                                "LOCphy" : 92,
                                "LOCpro" : 57,
                                "LOCpro_alarmed" : false,
                                "LOCbl" : 0,
                                "LOCcom" : 35,
                                "LOCcom_alarmed" : false,
                                "N" : 199,
                                "N1" : 107,
                                "N2" : 92,
                                "n" : 44,
                                "n1" : 22,
                                "n2" : 22,
                                "V" :1086.427,
                                "V_alarmed" : false,
                                "B" :0.452,
                                "D" :46.000,
                                "E" :49975.637,
                                "L" :0.022,
                                "T": "00:46:16" ,
                                "MaxND" : 3,
                                "MIwoc" : 91,
                                "MIcwc" : 41,
                                "MI" : 131,
                                "MI_alarmed" : false
                        }
                }
                ],

                "file_total" : {
                        "vG_b" : 6,
                        "vG_e" : 7,
                        "LOCphy" : 92,
                        "LOCbl" : 0,
                        "LOCpro" : 57,
                        "LOCcom" : 35,
                        "semicolons" : 26
                }
        },
        {
          "name": "StackTest.java" ,
... etc., similar information of other functions, until there comes
        "system" : {
                "packages" : 0,
                "files" : 2,
                "measured_items" : 7,
                "LOCphy" : 109,
                "LOCpro" : 73,
                "LOCcom" : 35,
                "LOCbl" : 1,
                "semicolons" : 35,
                "java_comment_blocks" : 8,
                "vG_b" : 7,
                "vG_e" : 8,
                "MIwoc" : 94,
                "MIcw" : 39,
                "MI" : 133,
                "methods_on_two_top_levels" : {
                        "total" : 5,
                        "LOCpro_avg" : 12,
                        "comment_percent_avg" : 33,
                        "vG_b_avg" : 2,
                        "vG_e_avg" : 2
                },
                "classes": {
                        "total" : 2,
                        "on_top_level" : 2,
                        "extends" : 0,
                        "implements" : 0
                },
                "interfaces": {
                        "total" : 0,
                        "on_top_level" : 0,
                        "extends" : 0,
                        "implements" : 0
                },
                "alarms" : {
                        "vG_alarmed" : 0,
                        "LOCpro_alarmed" : 0,
                        "comment_percent_alarmed" : 2,
                        "V_alarmed" : 0,
                        "B_alarmed" : 0,
                        "MI_alarmed" : 0
                },
                "alarm_limits" : {
                        "top_level_classes" : {
                                "vG_low" : 1,
                                "vG_high" : 100,
                                "LOCpro_low" : 4,
                                "LOCpro_high" : 400,
                                "comment_percent_low" : 20,
                                "comment_percent_high" : 60,
                                "V_low" : 100,
                                "V_high" : 8000,
                                "B_low" : 0,
                                "B_high" : 2,
                                "MI_low" : 80
                        },
                        "second_level_classes" : {
                                "vG_low" : 1,
                                "vG_high" : 100,
                                "LOCpro_low" : 4,
                                "LOCpro_high" : 400,
                                "comment_percent_low" : 20,
                                "comment_percent_high" : 60,
                                "V_low" : 100,
                                "V_high" : 8000,
                                "B_low" : 0,
                                "B_high" : 2,
                                "MI_low" : 80
                        },
                        "methods" : {
                                "vG_low" : 1,
                                "vG_high" : 10,
                                "LOCpro_low" : 1,
                                "LOCpro_high" : 40,
                                "comment_percent_low" : 20,
                                "comment_percent_high" : 60,
                                "V_low" : 4,
                                "V_high" : 1000,
                                "MI_low" : 80
                        }
                },
                "error_messages" : {
                        "count" : 0,
                        "msg" : [
                        ]
                }
        }
}
```

The JSON tags should be rather self-explanatory, but some remarks of them:

- In report header `"license_notes":[...]`: If the `cmtjava.ini` configuration file has such `NOTEn` lines, they are copied here. Evaluation copy license normally has such lines.
- `"date":"...",`: is the date when the cmtjava run was done.
- `"cwd":"..."`: is the current working directory, where the cmtjava run was done. It is useful in determining the source files, if they are specified with relative names.
- `"file":[\{... \` , { ...}] the array of files measured
- In `"stamp=...` the stamp is file's last modification time, the one which shows e.g. in '`dir`' listings. It is useful when wanting to check that the file is unchanged since it was measured.
- In various places there are two McCabe cyclomatic numbers,

  `vG_b` and `vG_e`. They stand for basic and extended. As you recall CMTJava can calculate this measure in four flavors: `basic`, `extended`, `basic_modified`, `extended_modified`. McCabe `basic[_modified]` cyclomatic number is shown in `vG_b` and McCabe `extended [_modified]` cyclomatic number is shown in `vG_e`. There is no distinction whether the measure is `_basic` or not. That information can be seen from the `"mccabe_preference ":...` tag.

If the report had been produced with `-jf` (long with frequencies) option instead on `-j` only, there would have been information of operand and operators per each method and class/interface. For example of the method `Stack.clear()`, it would have been:

```
''operators'' : {
  ''()'' : 1,
  '';'' : 3,
  ''='' : 3,
  ''public'' : 1,
  ''{}'' : 1    },
''operands'' : {
  ''0'' : 2,
  ''Stack'' : 1,
  ''myheight'' : 1,
  ''null'' : 1,
  ''size'' : 1,
  ''value'' : 1 }
```

The Excel output could have been produced as follows:

```
C:\TESTDIR> cmtjava -C EXCEL_FIELD_SEPARATOR=; -x -o xreport.txt
                  Stack.java StackTest.java
```

and the `xreport.txt` report file would be as follows:

```
File;Line;Measured_item;vG_b;vG_e;Params;MaxND;LOCphy;LOCbl;LOCpro;LOCcom;V;B(x100);T;N1;N2
    ;n1;n2;D;E;L(x1000);MIwoc;MIcw;MI;WarnMask
"Stack.java";14;".Stack.Stack()
    ";1;1;0;1;9;0;6;3;55;1;00:00:08;9;7;5;6;3;161;343;114;39;153;0
"Stack.java";25;".Stack.clear()
    ";1;1;0;1;7;0;4;3;29;1;00:00:03;5;4;5;4;3;71;400;122;42;164;0
"Stack.java";34;".Stack.getHeight()
    ";1;1;0;1;9;0;4;5;24;1;00:00:03;5;3;5;3;3;60;400;119;46;164;0
"Stack.java";44;".Stack.push()
    ";3;3;1;2;18;0;13;5;260;8;00:03:31;29;27;13;12;15;3803;68;95;36;131;0
"Stack.java";63;".Stack.pop()
    ";2;2;0;2;17;2;10;5;127;4;00:01:10;17;14;10;7;10;1267;100;99;37;137;0
"Stack.java";81;".Stack.top()
    ";2;2;0;1;14;2;7;5;104;3;00:00:43;13;12;10;8;8;782;133;104;40;144;0
"Stack.java";96;".Stack.getElement()
    ";2;3;1;1;14;2;6;6;125;4;00:01:03;14;15;11;9;9;1149;109;102;42;145;0
"StackTest.java";3;".StackTest.main()
    ";2;2;1;2;13;0;13;0;475;11;00:05:40;52;43;12;20;13;6128;78;97;0;97;100
```

Here, for printing reasons, the 9 rows are displayed on two lines. Also we had overridden the excel field separator to '`;`' while it by default in the configuration file settings is a '`\t`' (tab) character.

The first row specified the column headers and the next rows have the column values.

The "`Measured item`" column contains the method name (or internal 3rd level class/interface name). It is constructed to have "`package_name.class_name.method_name()`" ("`...class_name{}`"). If the source file does not have a package clause, i.e. the code belongs to an anonymous package, the `package_name` portion is empty, as in this example. If the measures item is an inner class or interface, there are "`{}`" instead of "`()`".

The "`Line`" column contains the line number where the measured item starts in its source file.

The last field on each row is "`WarnMask`". It contains an encoding on what measures were warned at the measured item. The encoding is represented as an integral value as follows:

0	No warnings
Add 1	McCabe warned
Add 10	LOCpro warned
Add 100	Comment percent warned
Add 1000	Halstead V warned
Add 10000	Halstead B warned
Add 100000	Maintainability Index warned

For example, `WarnMask` value `10010` would mean that Halstead *B* and *LOCpro* are warned (are outside of the acceptable ranges).

Note that this report form (when `-x` option and no `-s` option) has no top level or second level classes and interfaces.

With command

```
C:\TESTDIR> cmtjava -x -nxh -C EXCEL_FIELD_SEPARATOR=;
             -s -o xsreport.txt Stack.java StackTest.java
```

the following Excel data file `xsreport.txt` would result:

```
"Stack.java";1;".Stack
    {}";6;7;2;2;111;19;57;35;1086;45;00:46:16;107;92;22;22;46;49976;22;100;38;138;0
"StackTest.java";1;".StackTest
    {}";2;2;1;2;16;0;16;0;504;12;00:06:21;55;44;13;21;14;6859;73;93;0;93;100
```

Here you have only two rows. The column header row has been denied with `-nxh` option. In this report form the data rows are the measures of the top level classes and interfaces.

In the Excel reports note, that when the *B* value is represented as multiplied with 100 and the *L* value multiplied with 1000, none of the numerical values have a decimal separator. When also the values are separated by a semicolon or by a '`\t`' (tab) character (vs., say, with a comma), you can directly take this file as input to your Excel regardless of what your Excel assumes as a decimal separator (comma or period).

5.7. Using cmtjava2html Utility

The *cmtjava2html* utility is used to convert a short report (the default report form) into a hierarchical color-coded html representation, which can be viewed with commonly used html browsers. It is a command line utility, effectively a Perl script, which has the following command line options:

`-h` a command line help of the options.

`-i inputfile` Specifies the input file that is read by the tool. The input file must be a CMTJava Complexity Measures Report, so called short form, such that has been produced with *cmtjava* and when no `--l`, `-lf`, `-x`, `-s` options have been applied. When no `-i` option is given, *stdin* is read.

`-o output-dir` (optional) specifies the directory, which will be created if needed, and where to the HTML report is written. Default output directory is `CMTJAVAHTML` directory in current directory.

`-s source-dir` (optional) Specifies additional directories, from where cmtjava2html searches for the original source files for making an HTML'lized copy of them or at least a link to them.

The input CMTJava report contains the source file names in the way as they were given to cmtjava tool. They are either relative to the directory where the cmtjava run was done or they are absolute. If you are running the cmtjava2html utility in some other directory, you need to advise the tool where to look for the source files.

First the source file is looked with the (path and) name as it shows in the input report file, and no -s options are consulted. Then, if the file is not found and -s options are given, two algorithms are tried: 1) The source file name (and its possible path) is catenated to the arguments of the -s options. If the file gets found by any such name, we are done. 2) If the file is still not found, the source file name is stripped off from its possible directory path part and the file is searched from the directories given in the -s options. There can be many -s options. If the source file is not found at all, the resultant HTML report does not contain any links to view the source file.

`-l` *splitsize* (optional) Specifies how big html chunks are generated for the detailed report html page. Whenever at the beginning of a new source file the detailed html page contains already over *splitsize* lines, a new html page is started and linked with *next* and *previous* links to its neighbors. Splitting the detailed html report may be needed (for the sake of faster load times), if the html report contains much data (hundreds of source files, thousands of methods, ...). The default *splitsize* is at 1000 html lines.

`-p prefix` (optional) Determines how the generated html files are named in the output directory. They are named as `prefix.html`, `prefixA.html`, `prefixB.html`, etc. In the absence of this option the default for prefix is `index.` (Now, as of CMTJava v2.2, when there is `--o` option, you might consider this `--p` option as deprecated. It is however kept here for compatibility reasons.)

`-no-html-sources` (optional) Determines that no html'ized copies of the source Java files are generated in the output directory. Only links to the original source files are generated with such path and file names they could be resolved at the time and context of running *cmtjava2html*.

`-no-javascript` (optional) Determines that into the resultant HTML files there is not generated such html code that would require the use of Javascripts. (some people may have Javascripts disabled in their browser ...) The Javascripts are used to show the source code of a single method or class in the html report. If this option is given this functionality gets stripped off from the html report.

`-nsb` (optional) "Do Not Start Browser", advises *cmtjava2html* that browser is not started on the generated html representation, only the html files are generated. This option has effect only on Windows environment, where the default/automatic starting of the browser is only supported.

An example:

```
cmtjava -o report.txt Stack.java StackTest.java
cmtjava2html -i report.txt
start CMTJAVAHTML\index.html
```

The *cmtjava2html* utility does the following:

- Reads the CMTJava Complexity Measures Report that is given with the `--i` option. The input file must be of "short tabular form", i.e. one in whose generation it has not been used any of the options `-s, -x, -l, -lf`.
- Creates, if needed, into the current working directory `CMTJAVAHTML` subdirectory or output directory given with `--o` option and writes there (blindly overwrites any possible previous files with same name) a number of html files making up the html representation of the input file. Unless denied with `-no-html-sources` option, also the html'ized copies of the Java source files are constructed.

- The file where the browsing is started is `prefix.html` (by default index.html) at that directory. (Using different `prefix`es the same directory can contain many html representations.)

When the html representation is opened with a browser a *summary window* is shown first. It contains the class/interface-level summaries with a histogram of the percent of alarms in them. The data lines represent summaries of the two top-most classes and interface. Red color indicates that there have been alarms at the measured item.

The class/interface name is a link to the *detailed window*. Clicking on it opens a new window and positions it to the detailed CMTJava listing, to the beginning of the file where the clicked class/interface resides. The *detailed window* is effectively as the input file, i.e. the CMTJava Complexity Measures Report, but made to a color-coded html form. The *detailed window* may be in a number of smaller html chunks (in how many depends on the `-l` *splitsize* option value), which load faster and which are linked together with *next* and *previous* links.

In the *detailed window* the source file name is a link. By default it points to the html'ized source file copy inside the output directory, and shows the whole source file. If `-no-html-sources` option was used, it is a link to the actual source file, and it is up to the browser that you use how the Java source file is shown. If the source file was not found at cmtjava2html time, the source file name is a link to a page, which tells that source file was not found at *cmtjava2html* time.

In the *detailed window* the measured item names (classes and methods) are by default (when option `-no-javascript` was not given) links, too. Clicking on the class/method name brings to the screen the lines from the source file that make up the corresponding class or method. The lines vanish from the screen by clicking the name again.

This showing of the source code of the individual classes and methods is implemented by Javascripts in the generated html. If the html has been generated in the default way, but your browser does not allow Javascripts, you can experience various warning messages from the browser, and these links do not function at all or they do not work as intended (depending on the browser). When `-no-javascript` option is used, the class and method names are not links at all, the generated html does not contain Javascipts, and you do not get any complaints from your browser, if it has Javascripts disabled.

The html representation gives you a fast way to browse the CMTJava Complexity Measures Report at summary level, at detailed level and all the way at the source code level.

The *cmtjava2html* utility is effectively a Perl script. On Unixes it is assumed the Perl is available there already. On Windows platform the utility uses a Perl interpreter that comes along with the CMTJava delivery package (will reside at `%CMTJAVAHOME%\perl`).

6. Interpreting Complexity Measures

This chapter discusses how the measurement results of CMTJava can be applied. It is not possible to give absolute limits to acceptable values. You can configure CMTJava for project specific needs by changing the limit definitions in the configuration file.

6.1. Lines-of-Code Metrics

Lines-of-code metrics are the 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. CMTJava calculates the following lines-of-code metrics:

- *LOCphy*: number of physical lines
- *LOCbl*: number of blank lines
- *LOCpro*: number of program lines (actual Java code)
- *LOCcom*: number of comment lines

A line containing both program code and a comment is counted to both *LOCpro* and to *LOCcom*. A blank line inside a `/*... */` comment block is counted to comment lines.

The following recommendations are given for the lines-of-code measures. See the configuration file (`cmtjava.ini`) on what of these recommendations can be fine-tuned. CMTJava will mark an alarm in the case that the measured value is outside the recommended bounds.

Method length should not be longer than about 40 program lines. With some comments and empty lines it makes about one page.

Class length should be 4 to 400 program lines. The smallest entity that might reasonably make up a class has the class name line, the braces and something inside, about 4 lines. Classes longer than 400 program lines (10..40 reasonably sized methods) are usually too long to be understood as a whole.

At least 20 percent and at most 60 percent of a class or a method should be comments. If less than 20% are comments the measured item is either very trivial or poorly explained. If more than 60% are comments, the measured item is not a program but a document. However, note that comment ratio is considered (and possible warnings given) only if the item is of some reasonably size (`NO_COMMENT_WARNINGS_BELOW` configuration parameters).

6.2. Cyclomatic number

The McCabe Cyclomatic number $v(G)$ describes the complexity of the control flow of the program. In CMTJava v3.0 there came possibility to measure this in four flavors, as `basic`, `extended`, `basic_modified` and `extended_modified`. Read more from Appendix B. Here this measure is discussed assuming that it's flavor is *extended*, which was the only supported flavor in previous tool versions and which is the default now.

For a single method the McCabe Cyclomatic number $v(G)$ is 1 + the number of conditional branching points in the method. The greater the cyclomatic number is the more execution paths there are through the code, and the harder it is to understand. Note, that cyclomatic number is insensitive to complexity of data structures, data flows, and module interfaces.

The cyclomatic number of a method should be less than 10. If a method's cyclomatic number is 10, there are at least 10 (but probably more) execution paths through it. More than 10 paths are hard to identify and test. Methods containing one selection statement with many branches make up here an exception.

6.3. Maximum Nesting Depth

Somewhat related to $v(G)$ measure is the *MaxND* measure. It is the maximum nesting depth of `{}` braces in a method body. 5 is a reasonable upper limit for *MaxND*. *MaxND* is reported in Excel and XML form reports.

6.4. Number of Method Parameters

Number of method parameters is calculated and reported in Excel and XML form reports.

6.5. Volume (V)

Halstead's volume V describes the size of the implementation of an algorithm. The computation of V is based on the number of operations performed and operands handled in the algorithm. Therefore V is less sensitive to code layout than the lines-of-code measures.

The volume V of a method should be at most 1000. A volume greater than 1000 tells that the method probably does too many things.

The volume V of a class should be at most 8000. The limits of the volume can be used for double-checking.

6.6. Estimate for Delivered Bugs (B)

Halstead's delivered bugs B is an estimate for the number of errors in the implementation.

Some authors feel that the calculated B value (how Halstead has initially defined it, see Appendix B "How the Measures Are Calculated") is too low. And if it were multiplied with a certain factor, a better estimate would result.

The configuration parameter `B_CORRECTION_FACTOR` . gives you means to apply that schema. The default value for this setting is 1.0, when there is no difference.

Delivered bugs in a class should be less than 2. The number of defects there really are tends to grow more rapidly than *B*.

6.7. Maintainability Index (*MI*/*MIwoc*)

Maintainability Index is calculated with certain formulae (see Appendix B “How the Measures Are Calculated”) from lines-of-code measures, McCabe measure and Halstead measures. *MI* is a composite measure, which strives to express the relative maintainability of a complete software system in a single number, which is straightforward to calculate and which would have good predictive value.

There are two variants of Maintainability Index: one that contains comments (*MI*) and one that does not contain comments (*MIwoc*). CMTJava calculates them both. Which one is shown in some of the CMTJava reports is selected by the configuration file setting `MI_PREFERENCE`.

CMTJava calculates Maintainability Index for methods, classes and interfaces (two top levels) and for the whole software.

Whether you follow *MI* or *MIwoc* depends on how much you trust on the validity of the commenting in the code. If the comments are very out-of-date or if they are just some standard header blocks with no real value to the maintainer, you might want to follow the *MIwoc* measure instead of *MI*. And, of course, with this MI measure and with all the other CMTJava measures, you should use your common sense and make some experimental measurements with some familiar software for finding the best-suited alarm limits.

6.8. Complexity, Quality Assurance, and Testing

The more complex a module is, the more likely it is to contain errors, and the more difficult it is to test. The more complex it is, the harder it is to maintain. The more it contains errors, the more it needs to be maintained. The problem feeds itself! Some algorithms are complex, regardless of the way they are implemented, but in most cases complex implementation is due to bad design.

Source code quality assurance usually utilizes teamwork methods like code reading, structured walkthroughs, and inspections. These methods are invaluable; a tool can never find all those kinds of errors that a human inspection can. In addition to that, the team inspections also help the members of the team to learn from each other. However, there is usually too little time to inspect all the code carefully. In such a case, it is important to select the most important and most error-prone modules to the inspection.

Obviously, the modules that have high complexities need to be inspected most carefully. How high “high” is depends, of course, on your development process and quality criteria. The default alarm limits of CMTJava are one starting point. If you have collected information about error densities of your code, you can measure complexities of those codes and draw your own conclusions about suitable alarm limits.

When dynamic testing is concerned, the most important complexity measures are the cyclomatic number ($v(G)$) and the number of delivered bugs (B). Because the cyclomatic number describes the control flow complexity, it is obvious that classes and methods having high cyclomatic number need more test cases than modules having a lower cyclomatic number. As a rule of thumb, each method should have at least as many test cases as indicated by its cyclomatic number. 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.

A. The Source Code Language

This appendix discusses how CMTJava analyzes the source language.

CMTJava operates on Java. The tool does not actually check the input file syntax for being correct but assumes so. If it is syntactically incorrect CMTJava may not be able to recognize the input, especially if the "top-level" program structures are syntactically incorrect. When the tool parses and calculates for example method bodies, i.e. the code inside `{...}`, there may be syntactically invalid constructs that may get accepted by the tool. One thing, which the tool does not understand well, however, is such an erroneous program, which has an unbalanced pair of `{}`, `[]`, or `()` parenthesis.

CMTJava recognizes and processes `/*... */` block comments and `//...` line comments. The number of `/** ... */` comment blocks, so called Java comment blocks, are recognized and reported in short tabular form report over all files.

B. How the Measures Are Calculated

B.1. Lines of code Metrics

The lines of code measures are calculated according to the following rules:

- Blank lines contain only whitespace characters.
- Comment lines are the lines containing a comment or a part of a comment. A blank line inside a `/*... */` block comment is counted to comment lines.
- Program lines contain program code, other than comments.

Comments and program code may share the same physical line. Such lines increase both the number of comment lines and the number of program lines.

LOCphy for a file is the number of physical lines in that file. *LOCphy* is equal to the number of newline characters in the file, or that $amount + 1$, if the file does not end with a newline. For a class, interface and method *LOCphy* is the number of lines that the construct occupies.

LOCbl is the number of blank lines. *LOCcom* is the number of lines containing a comment or a part of a multi-line comment. *LOCpro* is the number of lines containing Java tokens.

Configuration file setting `NOTICE_LEADING_COMMENTS=1` determines whether a block of comment lines (which may be followed by blank lines), which immediately precede a class, interface or method, is counted to the lines of measured item at hand.

B.2. Halstead Metrics

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 it is counted

- **number of unique operators** (*n1*)
- **number of unique operands** (*n2*)
- **total number of operators** (*N1*)
- and **total number of operands** (*N2*)

in the piece of source code under measurement (class, interface, method). Other Halstead measures are derived from these with certain fixed formulas as described later.

The following tokens are counted to **operands**:

- All identifiers that are not keywords.

- Keywords that are predefined type names `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.
- Keywords `true`, `false`, `null`, `super`, `this`, `void`.
- Numeric and string literals

All the remaining tokens are counted to **operators**. However, note

- `:` `:`' in the construct "`case ... :`" is not counted separately but considered to be part of the construct
- '`(...)`' in "`if(...)`", "`for(...)`", "`while(...)`", "`switch (...)`", "`catch(...)`" is not counted separately but considered to be part of the construct
- the pair of parentheses "`(...)`", "`{...}`", "`[...]`" are counted as a pair, not individually, and considered to be one operator.

Blank lines and comments are counted neither to operands nor to operators.

The number of unique operators and operands (*n1* and *n2*) as well as the total number of operators and operands (*N1* and *N2*) are calculated by collecting the frequencies of each operator and operand token of the source program.

All other Halstead's measures are derived from these four quantities using the following set of formulas.

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

$$N = N1 + N2$$

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

$$n = n1 + n2$$

The **program volume** (*V*) is the information contents of the program, measured in mathematical bits. It is calculated as the program length times the 2-base logarithm of the vocabulary size:

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

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 ratio 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).

$$D = \frac{n1}{2} \cdot \frac{N2}{n2}$$

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.

$$L = \frac{1}{D}$$

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

$$E = V \cdot D$$

The **time to implement** or understand a program (T) is proportional to the effort. Empirical experiments can be used for calibrating this quantity. Halstead has found that dividing the effort by 18 gives an approximation for the time in seconds.

$$T = \frac{E}{18}$$

The **number of delivered bugs** (B) correlates with the overall complexity of the software. Halstead gives the following formula for B:

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

This value is multiplied with the configuration parameter `B_CORRECTION_FACTOR` value and the result is displayed in the tool reports. When `B_CORRECTION_FACTOR=1.0`, the calculated B value (as defined by Halstead) and the displayed B value are the same.))

B.3. McCabe Metrics

McCabe Cyclomatic number $v(G)$ is calculated on methods, classes and interfaces. v(G) can be calculated in four flavors, as `basic`, `extended`, `basic_modified`, and `extended_modified`. The default is `extended` (`MCCABE_PREFERENCE` setting in `cmtjava.ini`).

Consider first this measure on methods:

McCabe's Cyclomatic number $v(G)$ shows the complexity of the flow of control through a piece of code. $v(G)$ is the number of conditional branches in the flowchart. Thus a method consisting of only sequential statements has $v(G) = 1$.

Each if-statement introduces a new branch to the program and therefore increases $v(G)$ by one. Iteration constructs for- and while-loops introduce branches, both adding one to $v(G)$.

Construction `expr1 ? expr2 : expr3` increases $v(G)$ by one.

Each `catch (...)` part in a try-block increases $v(G)$ by one. The possible `finally` part in a `try`-block does not increase $v(G)$, because it is always executed (not any conditional execution of it).

Difference between `basic[_modified]` and `extended[_modified]` is in how the && and `||` operators in conditional expressions are calculated to $v(G)$. Consider the following code snippet:

```
if ( a > 5 || (b >= 0 && b < 10)) { ...
```

In `basic[_modified]` $v(G)$ is added only with 1 (of the ‘`if`’) while in `extended[_modified ]` $v(G)$ is added with 3 (of ‘`if`’, ‘`||`’ and ‘`&&`’).

The difference between `basic` and `basic_modified` (similarly between the variants `extended` and `extended_modified`) flavors is in how the ‘`switch`’ line and the ‘`case n:`’ label lines are calculated.

When the flavor has not `_modified` property the ‘`switch`’ statement is calculated as follows:

- If the ‘`case n:`’ branch has executable code in it, it adds 1 to $v(G)$.
- If the ‘`case n:`’ branch has not executable code in it (it flows through to the next branch that has executable code in it), $v(G)$ is not increased.
- ‘`default:`’ branch, regardless if it has or has not executable code in it, does not increase $v(G)$. Whether the ‘`default:`’ is present or not, it does not increase the conditionally executed branches of the ‘`switch`’ statement.

When the flavor has *_modified* property the ‘switch’ statement is calculated as follows:

- The ‘`switch(...)`’ line adds 1 to $v(G)$.
- Of the ‘`case n:`’ branches and of the ‘`default:`’ branch nothing is added to $v(G)$.

The rationale for measuring $v(G)$ as `_modified` is that a ‘`switch`’ statement can have large number of ‘`case n:`’ branches, which can put the overall $v(G)$ value over the alarm limit. But often the ‘`case n:`’ branches are quite similar to each other and in that perspective the whole ‘`switch`’ could be considered as one conditional branch only.

Still an example. Consider two methods:

```
public int foo() {
    a++;
    return 5;
}

public int bar(int i, int j) {
    if (i > 10 || (j > 20 && j < 30)) {
        a++;
    } else if ( j > 0) {
        return 10;
    }
    switch (i + j) {
        case 1: a++; break;
        case 2: b++;
        case 3: c++; break;
        case 10:
        case 11:
        case 13:
        case 15: d++; break
        case 20: e++;
        default: f++;
    }
    return 20;
}
```

On method `foo()` $v(G)$ is 1 in all $v(G)$ flavors. On method `bar()`, when the flavor is

- `extended`: $v(G) \geq 10$, note: no $v(G)$ points of case `10, 11, 13, default` branches
- `basic`: $v(G) \geq 8$, note: no $v(G)$ points of '`||, &&`.
- `extended_modified`: $v(G) \geq 6$, note: no $v(G)$ points of `case n`'s, 1 $v(G)$ point of switch.
- `basic_modified`: $v(G) = \geq 4$. note: similar as `extended_modified`, except no $v(G)$ points of '`||, &&`.

It should be noted that $v(G)$ is insensitive to unconditional branches like `return`-, `break`- and `throw`-statements although they surely increase complexity.

Then consider how $v(G)$ is calculated on classes and interfaces:

The rules are the same regardless if the entity is a file-level class/interface or it is an inner class/interface to a file-level class/interface.

- Start calculations from 1.
- If the class/interface contains methods or inner class/interfaces, whose McCabe complexity is over one, add of each of them the amount how much they are over one. For example, if some method has complexity 5, the class/interface level complexity measure is increased by 4.
- If in a class/interface there is static initializing code, its $v(G)$ is calculated in a normal manner and it increases the $v(G)$ of the class/interface with its value.

So, for example, if a class has only some number of methods, that each have $v(G) = 1$, then for the class it is also $v(G) = 1$. The algorithmic complexity, for example conditional execution of the class methods, is somewhere else, outside the class itself.

At file level the $v(G)$ is calculated as follows:

- Start calculations from 1.
- Add the $v(G)$ of each top level class and interface to file's $v(G)$ and subtract 1. I.e. a class/interface with $v(G) = 1$ does not increase the file's $V(G)$.

At system level the $v(G)$ is calculated as follows:

- Start calculations from 1.
- Add the $v(G)$ of each file to system level $v(G)$ and subtract 1. I.e. a file with $v(G) = 1$ does not increase the system's $v(G)$.

B.4. Maximum nesting depth

Maximum nesting depth *MaxND* is calculated on methods. It is reported also on class level, where it means the maximum *MaxND* value of the class's methods.

MaxND is a measure on how deep is the maximum `{}` nesting in the method. For example, in the following code snippet

```
void foo1() {
.../* some code having no {}s */
 }
void foo2() {
   if(cond1) {
      if(cond2) {
         /* some code having no {}s */
      }
   }
   if(cond3) {
      /* some other code having no {}s */
   }
}
```

Here the *MaxND* is 1 for `foo1` and 3 for `foo2`. In the counting only the explicit `{}`s are noticed that are in the method body .

The *MaxND* is somewhat similar to $v(G)$, but gives another and supplementary view to the algorithmic complexity of the code.

B.5. Maintainability Index (*MI*)

Maintainability Index (*MI*) is calculated on each measured item and additionally on all files together ("system") level. Actually there are three measures:

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

The general formulae for MI is the following:

$$\begin{aligned}
MIwoc &= 171 - 5.2 \cdot \ln(aveV) - 0.23 \cdot aveG - 16.2 \cdot \ln(aveLOC) \\
MIcw &= 50 \cdot \sin(\sqrt{2.4 \cdot perCM}) \\
MI &= MIwoc + MIcw
\end{aligned}$$

Where

- $aveV$ = average Halstead Volume (V measure) per module
- $aveG$ = average extended cyclomatic complexity ($v(G)$ measure) per module

- *aveLOC* = average count of lines (*LOCphy* measure) per module

The above definition uses a concept ‘module’. The general MI literature explains the ‘module’ be “any named lexical component of a program, which given a specific programming language might be a function, a procedure, a subroutine, a section, a module, etc.”.

In CMTJava ‘module’ is taken to be a method. Also any third level class/interface is taken to be a ’module’. (Right or wrong, according to the CMTJava’s selected principle, the third level classes and interfaces are just thrown to the same category with methods. Such third level classes/interfaces are presumably rare.) MI for the top level and second level classes is calculated via their participating module averages. The system level MI is similarly calculated via the averages of all encountered modules.

C. cmtjava Error Messages

Each error message takes one of the following forms:

```
*** CMTJava error error-code:  While processing file file-name around line
                          line-number error-text
```

or

```
*** CMTJava fatal error error-code:  error-text
```

or

```
*** CMTJava warning error-code:  While processing file file-name around
                          line line-number error-text
```

where `error-code` is an integer value used to identify the error (helps in communicating with Testwell) and `error-text` is the actual error message. The message portion While processing file `file-name` around line `line-number` comes only if the message is related to processing of some file.

As a program the cmtjava returns an exit code to the operating system level. The exit codes are the following:

`0` cmtjava run ended normally with no error messages.

`1` cmtjava run ended normally but some error messages were written.

`2` cmtjava run ended to a fatal error and the run was aborted.

The cmtjava messages are written to *stderr*.
The possible `error-texts` are the following:

- `Bad character constant`
  Syntactically erroneous character constant.
- `Bad combination of options on command line, type -h to get help`
  For example there was -x and -l options at the same time.
- `Bad conf parameter 'xxx'` Configuration file contained a bad parameter definition.
- `Bad definition xxx in configuration file`
  Bad definition in configuration file
- `Bad string constant, ending " missing`
  Syntactically erroneous string constant.

- `Huge string literal, cutted`
  The string is longer than CMTJava is prepared for; only the string begin is reported.

- `Cannot create file` *`filename`*
  Creation of the given file for CMTJava run output failed.

- `Cannot open source file` *`filename`*
  The given source file could not be opened.

- `CMTJava run aborted`
  After searching all the locations for configuration parameters and noticing the possible `-c` option there still was one or more required configuration parameters unset. The previous message described the more detailed reason.

- `Could not find configuration file` *`filename`*
  The configuration file given explicitly in the `-c` option could not be opened, or, in the absence of `-c` option, none of the configuration files were found from default search locations.

- `Could not read/close file` *`filename`*
  Configuration file reading or closing had failed.

- `Internal error`
  CMTJava internal sanity checks for its behavior. Assuming your source is correct Java you should not get these.

- `License problem:` *`problem-description`*
  There is a problem in your CMTJava license as described in the problem-description, for example “copy protection module not found”, “the license has expired”, “IP address of this machine (*xx.xx.xx.xx*) is not listed in the COMPUTER field in the configuration file”, etc. (These license control routines are used in all Testwell C/C++ and Java test tools)

- `Out of memory`
  CMTJava detected a short of memory condition.

- `Syntax error: stray '\' in program`
  '\' did not start a legal escape sequence.

- `Syntax error: unterminated comment`
  Comment block starting with `/*...` ended with `EOF`.

- `There is a problem with software license`
  There is a problem with software license.

- `Syntax error, unexpected end-of-file`
  While looking for a specific token, normally one of '>', ')' or '}', and unexpected end of file was met.

- `Unexpected eof, '}' for class or interface expected.`
  Unexpected end of file in source file reading.
- `Syntax error:  "token" was met when class/interface/enum was expected, file end not measured.`
  Erroneous input file.
- `Unknown or badly placed option -`*`xxx`*`, type -h to get help`
  The option is not known to CMTJava or it is placed after the source file names on the command line.
- `Bad -C option 'xxx'`
  Error in the -C option argument 'xxx'.
- `Unrecognised input token`
  Are you sure that the source code is correct Java.
- `Unrecognised input token, ';' pretended`
  Are you sure that the source code is correct Java. Source file parsing continues as if there had been a ';'.
- `<Fatal error message related to configuration file handling>`
  These messages have error code 2. There are a number of possible messages of this category (used in all Testwell C/C++ and Java tools), like "missing license parameter: *xxx*", "TOOL mismatch", "wrong TOOL version", etc.

D. cmtjava2html Error Messages

The cmtjava2html error messages are written to *stderr* and they have the following form:

```
cmtjava2html: error message
```

where the *error message* can be one of:

- `Erroneous command line option: option usage`
  Bad option when invoking cmtjava2html.
- `Error input line: linenumber`
  The input file seems to be not of the CMTJava report type that cmtjava2html assumes as input, detected at the given line.
- `File output-dir exists and is not a directory`
  Subdirectory output-dir could not be created. Rename the file to another name.
- `Can not open file filename for writing: op_syst_error_text`
  For some reason this operation failed.
- `Can not open file filename for reading: op_syst_error_text`
  For some reason this operation failed.
- `Unexpected end of file: inputfilename`
  The input file ended unexpectedly

Index