Rational. Logiscope
Release 6.6

IBM

Basic Concepts

Before using this information, be sure to read the general information under "Notices" section, on page **151**.

About this manual

Audience

This manual is intended for people who wish to find out on the key concepts of IBM® Rational® Logiscope™ . This manual should be read before the others.

Overview

This manual contains the following chapters:

Chapter 1 describes the Logiscope environment, the modules, the toolset and the main input and output data.

Chapter 2 provides information on *Call Graphs.*

Chapter 3 provides information on *Control Graphs.*

Chapter 4 explains how to use *Logiscope QualityChecker* for evaluating software quality using source code metrics.

Chapter 5 introduces the main standard metrics available with *Logiscope QualityChecker.*

Chapter 6 describes the structure and syntax of the Logiscope Quality Model file and Rule Set file.

The manual ends with a bibliography and a glossary on Logiscope concepts.

Conventions

The following writing conventions are used in this manual:

- **bold**: names of commands (**call**) and file extensions (**.res**)
- *italic*: names of user-defined textual elements (*version_1, component_2*), notes,
- `typewriter`: screen messages (`Reference filename`) requiring user action,
- keycaps (<Enter>).

Contacting IBM Rational Software Support

If the self-help resources have not provided a resolution to your problem, you can contact IBM® Rational® Software Support for assistance in resolving product issues.

Note If you are a heritage Telelogic customer, you can go to http://support.telelogic.com/toolbar and download the IBM Rational Telelogic Software Support browser toolbar. This toolbar helps simplify the transition to the IBM Rational Telelogic product online resources. Also, a single reference site for all IBM Rational Telelogic support resources is located at http://www.ibm.com/software/rational/support/telelogic/

Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html

• To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/ brochures_faqs _quickguides. html.

• For further assisance, contact your IBM representative

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

• Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to http://www-01.ibm.com/software/support/..

• Be listed as an authorized caller in the service request tool

Submitting problems

To submit your problem to IBM Rational Software Support:

1. Determine the business impact of your problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

Use the following table to determine the severity level.

Severity	Description
1	The problem has a *critical* business impact. You are unable to use the program, resulting in a critical impact on operation. This condition requires an immediate solution.
2	The problem has a *significantl* business impact. The program is usable, but it is severely limited.
3	The problem has a *some* business impact.The program is usable, but less significant features (not critical to operation) are unavailable.
4	The problem has a *minimal* business impact.The problem causes little impact on operations or a reasonnable circumvention to the problem was implemented.

2. Describe your problem and gather background information, When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?

  To determine the exact product name and version, use the option applicable to you:

  - Start the IBM Installation Manager and select **File** > **View Installed Packages**. Expand a package group and select a package to see the package name and version number.
  - Start your product, and click **Help** > **About** to see the offering name and version number.
- What is your operating system and version number (including any service packs or patches)?
- Do you have logs, traces, and messages that are related to the problem symptoms?
- Can you recreate the problem? If so, what steps do you perform to recreate the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
- Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

3. Submit your problem to IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:

- **Online**: Go to the IBM Rational Software Support Web site at https://www.ibm.com/software/rational/support/ and in the Rational support task navigator, click **Open Service Request.** Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.

  For more information about opening a service request, go to http://www.ibm.com/software/support/help.html

  You can also open an online service request using the IBM Support Assistant. For more information, go to http://www-01.ibm.com/software/support/isa/faq.html.

- **By phone**: For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at http://www.ibm.com/planetwide/ and click the name of your country or geographic region.

- **Through your IBM Representative**: If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at http://www.ibm.com/planetwide/.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Rational Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Rational Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Rational Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Table of Contents

Chapter 6   **Project Configuration Files**

Chapter 7   **Bibliography**

Chapter 8   **Terms and Definitions**

Chapter 9   **Notices**

The Logiscope Environment

1.1 Life-cycle Processes

The quality of a software product is constructed throughout its life-cycle.
Quality requirements should be expressed and defined during the earliest phases of development. They should be taken into account and verified in all subsequent phases.
Rational Logiscope is a toolset used to verify that quality requirements have been fulfilled and maintained all along the software life cycle.
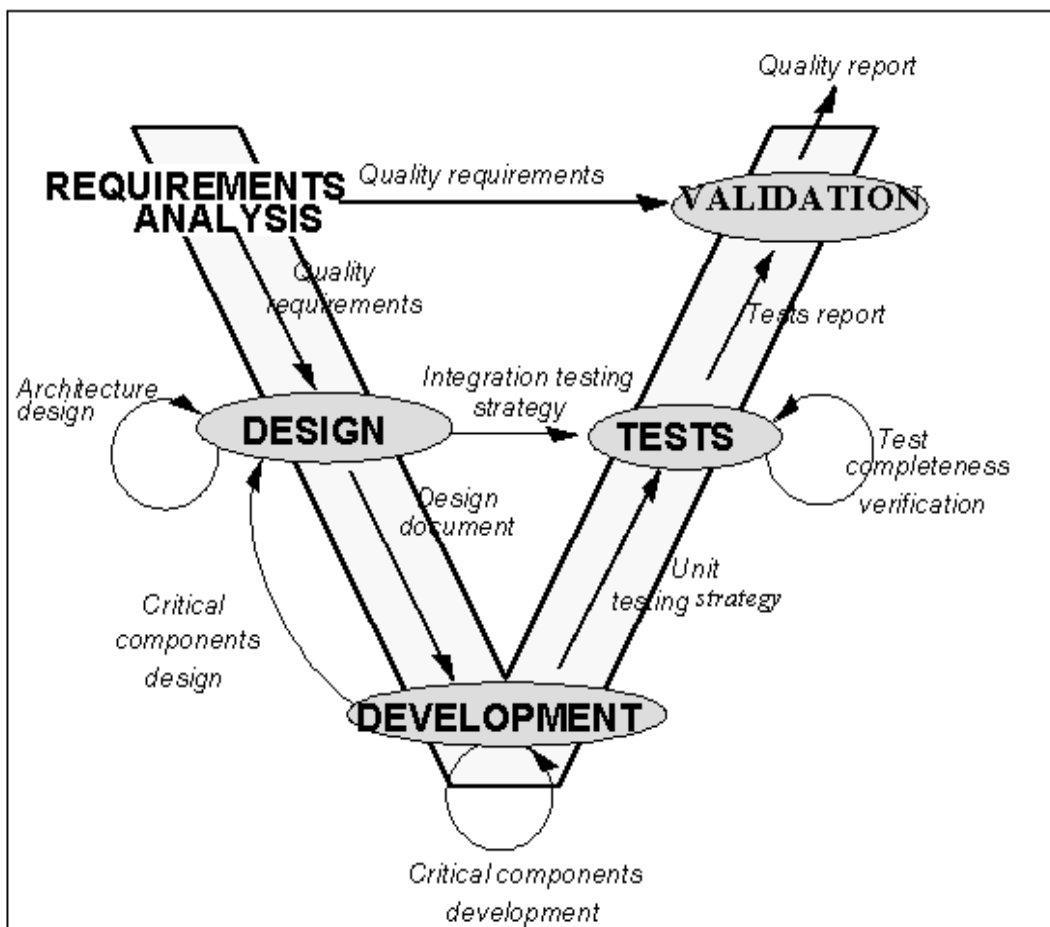
Quality requirements mentioned during the specification phase are formalized during design and development phases.

1.1.1 Design and Development

Using *Rational Logiscope* during design and development phases allows verification of the quality of the software architecture and coding. It makes it possible to detect critical elements at the earliest possible stages and to act before corrective and maintenance efforts become prohibitive.

Developers and designers are becoming aware of the importance of mastering quality in the same way that the development of a function must be mastered. With *Rational Logiscope*, users learn as they advance through the project.

The documentation drawn from these phases constitutes the reference elements for subsequent phases and for the software maintenance period. *Rational Logiscope* helps to draft standardized in-house documentation for improved communication between different groups involved in the project.

Test strategies are defined while the software is being constructed. *Rational Logiscope* helps to draw up these strategies and to adapt the test effort to software complexity.

1.1.2 Testing

Testing is more efficient with *Rational Logiscope* which allow measuring the completeness of test plan with respect to software structure. The objective assessment of test efficiency in terms of paths covered makes it possible to ensure that a satisfactory level of testing has been reached. When used by testers, it helps to define complementary tests in order to guarantee a satisfactory level of reliability. In particular, it can be used to draft test reports.

1.1.3 Verification & Validation (V&V)

V&V must ensure that all features defined in the specification are fulfilled. Among other things, validation must verify that quality requirements are met. *Rational Logiscope* measures the product's intrinsic characteristics (analysability, changeability, testability, etc.) and the test completeness. These measurements can be compared with standards defined at the start of the project in order to meet quality requirements.

1.1.4 Maintenance

Maintenance alone could be considered as a software production cycle except that it is based on existing software.

If the existing software is a program whose quality has been assured, maintenance will take place naturally, the only preoccupation being not to introduce regression. This can easily be verified by using *Rational Logiscope*.

However, if the state of the existing software is uncertain, a study will have to be carried out in order to plan the maintenance effort. *Rational Logiscope* can rapidly propose an evaluation of the software quality.

1.1.5 Project Management

With concise results allowing quality to be assessed, project management itself monitors the progress made in the project.

1.2 The Logiscope Verification Modules

Rational Logiscope offers several complementary verification modules.

- *Rational Logiscope QualityChecker* uses source code metrics to locates complex, error-prone modules, and provides graphic results to assess architecture of the application and detailed design. *Rational Logiscope QualityChecker* helps you diagnose problems and improves your decision-making — should you rewrite the module or test it more thoroughly? Software metrics Quality Modeling is ISO 9126 compliant and configurable to project-specific requirements.

- *Rational Logiscope RuleChecker* automatically checks your code against a set of project-defined programming rules in order to avoid language traps and code misunderstandings.
  Rational Logiscope RuleChecker comes with a configurable set of coding and naming rules including MISRA C and C++ coding rules (cf. [MISRA-C:1998], [MISRA-C:2004], [MISRA-C++:2008]) .

- *Rational Logiscope CodeReducer* automatically identifies similar pieces of code. It can be used in many situations:
    - searching for all similar pieces of code in a given set of files
    - searching for code similar to a reference code in a given set of files,
    - comparing source code files of distinct variantes of 2 projects,
    - searching for differences between two versions of a set of source code files.

  In all these cases, *Rational Logiscope CodeReducer* helps you factorizing and reducing your source code size.

- *Rational Logiscope TestChecker* supports structure-based testing and test coverage analysis (cf. [DO178B]. It quantifies test completeness and shows uncovered source code paths. By uncovering bugs hidden in untested source code, *Rational Logiscope CodeReducer* improves program reliability.
  Rational Logiscope TestChecker is based on source code instrumentation techniques that are adaptable to your test environment constraints, both on host and target platforms.

1.3 The Logiscope Languages

All the verification modules mentionned in the previous section are available for Ada, C, C++ and Java source code.

In the Logiscope documentation, <Language> designates Ada, C, C++ or Java.

1.4 The Logiscope Project and Workspace.

Logiscope uses "projects" to manage data. A Logiscope project ("**.ttp**") mainly consists in:

• the list of source files to be analysed,

• applicable source code parsing options according to the compilation environment,

• the verification modules to be activated on the source code files and the associated controls (e.g. metrics to be computed, rules to be checked).

One or more projects can be included in a Logiscope workspace ("**.ttw**").

A source file is a file containing source code. This file is not necessarily compilable. It only has to conform to the language syntax.

1.5 The Logiscope Toolset

Logiscope is composed of a set of tools. The main tools are the following:

1.5.1 Logiscope Graphical User Interface

• **Logiscope Studio** allows to create Logiscope project, "build" projects (i.e. launch source code parsers to extract all necessary information), visualize the results (e.g. browse within the code to locate the violations of a coding standard) produced by the selected verification modules and generate reports in various formats (e.g. HTML, CSV).
For more detail, please refer to *IBM Rational Logiscope QualityChecker & RuleChecker Getting Started* and *IBM Rational Logiscope Studio Reference Manual*.

• **Logiscope Viewer** mainly provides graphical results to assess architecture of the application and detailed design:

- Call graph: a graphical representation of the calling relationship between the functions defined in the application; for more detail, please refer to chapter 2.

- Use graph: a graphical representation of the use relationship between the classes defined in the application (only applicable to C+ and Java),
- Control graph: a graphical representation of the chaining and nesting of control structures within a function; for more detail, please refer to chapter 3.

In addition, the **Logiscope Viewer** provides many results for assessing quality using source code metrics: e.g kiviat graph. For more detail, please refer to chapter 4.

- **Logiscope TestChecker** runs test cases and computes test coverage:
  For more detail, please refer to *IBM Rational Logiscope TestChecker Getting Started*.

- **Logiscope Windoc** is only available on Windows platform. It allows to automatically generate Quality and Test reports in MS Word format.
  For more detail, please refer to *IBM Rational Logiscope RuleChecker and QualityChecker Getting Started*.

1.5.2 Logiscope Command Line Mode

- **Logiscope create** is a tool to be used from a standalone command line or within makefiles to create Logiscope projects.

- **Logiscope batch** is a tool to be used from a standalone command line to build Logiscope projects and automatically generate verification reports.

- **Logiscope lgdynld** is a tool designed to merge test coverage files into one a single file.

1.6 The Logiscope Reference

The Logiscope Reference is a directory containing configuration files used by the Logiscope toolset according to the applicables verification modules:

- Rule Sets and associated rule scripts used by *Rational Logiscope RuleChecker*, for more details see section 1.6.1,

- the Quality Model and associated source code metrics used by *Rational Logiscope QualityChecker*, for more details see section 1.6.2,

- Contexts and Includes containing various internal scripts used in some other configuration files for more details see section 1.6.3.

A default Logiscope Reference environment is available in the **Ref** subdirectory of the standard Logiscope installation directory. The user can either modify the content of the standard Logiscope Reference directory or define a dedicated one in order to be tailored to the project or organisation constraints and quality objectives. See Logiscope Environment variables.

1.6.1 Rule Sets and Rules

A Rule Set is user-accessible textual file containing the specification of the programming rules to be checked on the application source files by *Logiscope RuleChecker*. Default Rule Sets are provided in the `RuleSets` folder of the default Logiscope Reference directory.
Specifying one or more Rule Sets is mandatory when setting up a *Logiscope RuleChecker* project.

The Rule Sets allow to tailor *Logiscope RuleChecker* verification to specific requirements: i.e. taking into the applicable coding standards.

* Rule checking can be activated or de-activated.

* Some rules have parameters that allow to customize the verification. Changing the parameters changes the behaviour of the rule checking.

* The default name of a standard rule can be changed to fit to the name and/or identifier specified in the applicable programming guide.
  The same standard rule can even be used twice with different names and different parameters.

* The default severity level of a rule can be modified.

* A new set of severity levels with a specific ordering: e.g. "Mandatory", "Highly recommended", "Recommended" can be specified.

All these actions can be done by editing the Logiscope Rule Set and changing the corresponding specification. The structure of a Rule Set file is detailed in section 6.2.

If you did not choose to generate a flat rule set when you created the project you may have to edit the Rule Set file that is included in your project's Rule Set file.

More information on how Rule Sets are set up in Logiscope projects can be found in the *IBM Rational QualityChecker & RuleChecker - Getting Started* manual.

1.6.2 Quality Model and Metrics

The Quality Model or Reference file is a user-accessible textual file containing the definition of the Quality Model required to assess quality characteristics such as: Maintainability, Reliability, Portability, etc. using source code metrics. Default Quality Model files are provided in the default Logiscope Reference directory.
Specifying a Quality Model file is mandatory when setting up a Logiscope *QualityChecker* project.

In order to adapt the Quality Model to the project constraints and quality objectives, the user can use the Quality Model Editor available in Logiscope **Studio.** The user can also directly edit the Quality Model file. In such a case, the user shall respect the structure and syntax described in section 6.1.

More information on how Quality Models are set up in Logiscope projects can be found in the Logiscope *QualityChecker & RuleChecker - Getting Started* manual.

1.6.3 Contexts and Includes

Contexts are calculated over the entire application before rules and metrics are, and may be used to store useful application information to be used in rules and metrics.

Contexts can be added to any type of project, metrics to *Logiscope QualityChecker* projects, and rules to *Logiscope RuleChecker* projects.

As the contexts exist to provide information to specific rules and metrics, their results do not appear as a given result in the Logiscope toolset.

For more details, please refer to *IBM Rational Logiscope RuleChecker - Writing Ada, C++ and Java scriptable rules, metrics and contexts.*

1.7   The Logiscope Repository

The Logiscope Repository is the directory where *Rational Logiscope* will create and maintain all internal files storing the necessary information.

At the end of the Logiscope project creation process, the following files are generated in the Logiscope Repository:

- <ProjectName>**.ttp** for Logiscope project,

- <ProjectName>**.ttw** for Logiscope workspace (once the project has been open by Logiscope **Studio**),

- <ProjectName>**.rst** for Logiscope Rule Set (if the *RuleChecker* module has been activated).

where <ProjectName> is the name of the Logiscope project.

Once the Logiscope project has been "built": i.e. the source files of the project have been parsed to extract all necessary information for code verification, a Logiscope folder is created containing several Logiscope internal files in ASCII format.

All files stored in the Logiscope Repository are internal data files to be used by *Rational Logiscope*. They are not intended to be directly used by the end-users. The format of these files is clearly subject to changes.

1.8 The Logiscope Environment Variables

If set, the environment variables for Logiscope are as follows:

- `LOG_REF_ENV`: designates the directories where the Logiscope Reference environnement are available.
  The syntax of `LOG_REF_ENV` is:
    - dir1;dir2;…;dirn (directory names separated by semi-colons) on Windows and,
    - dir1:dir2:…:dirn (directory names separated by colons) on Unix and Linux.

  Directories in `LOG_REF_ENV` shall have the same structure as the standard Logiscope Reference: i.e. the **Ref** sub-directory in the standard Logiscope installation directory: i.e. including sub-directories such as `"RuleSets/<Language>"` and `"Rules/<Language>"`.

- `LOG_ADDINS_ENV`: designates the directories where Logiscope **Studio** shall search for addins.

- `LOG_UTIL`: designates the **util** directory in the standard Logiscope installation directory.

Chapter 2

The Call Graph

2.1   Presentation

The call graph shows the overall architecture of an application by displaying calling relations between components of the application. For Ada, C++ and Java projects, the call graph is an estimated call graph (no evaluation of parameters type).

In this type of graph, nodes symbolize components (functions, methods, procedures, ...) and edges symbolize calling relationships between components. Calling relations are read from top to bottom (except for recursive calls). Below is an example of a simple call graph.
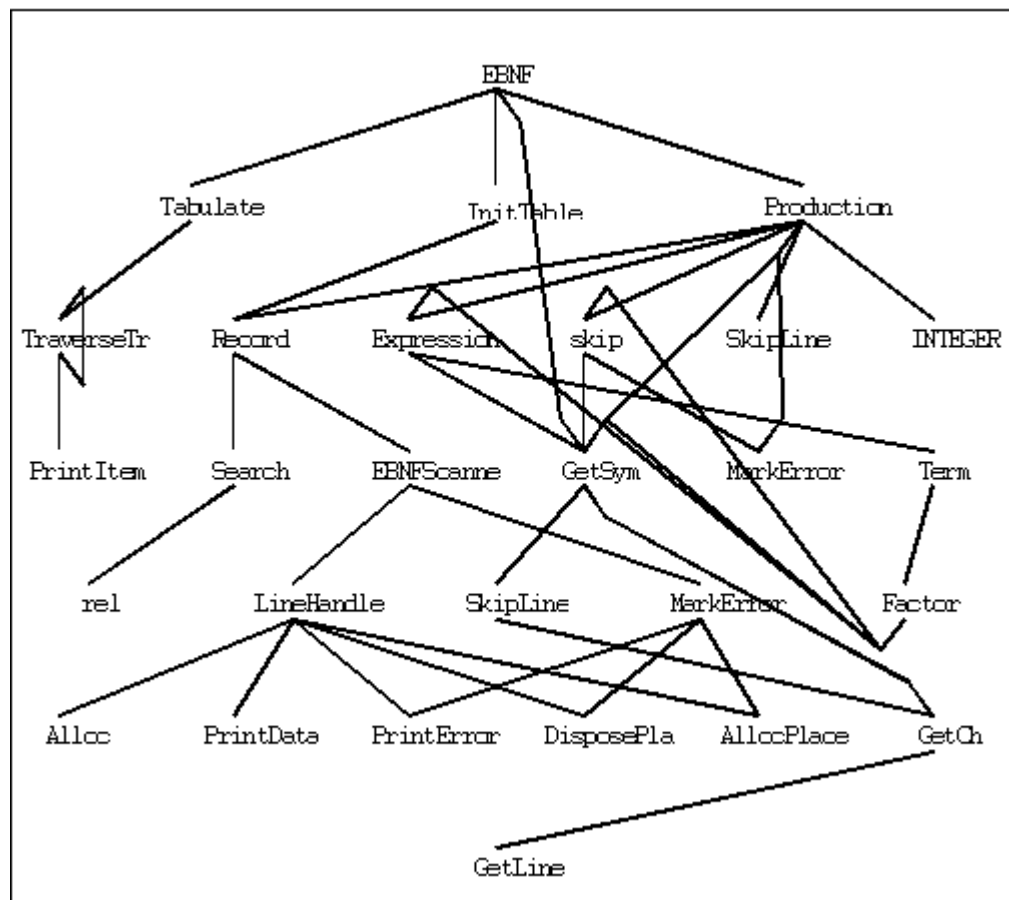
There are several types of calling relationships:

– simple call
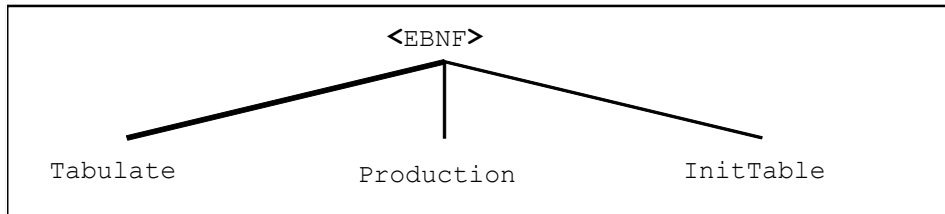
EBNF component calls the *Tabulate* component.
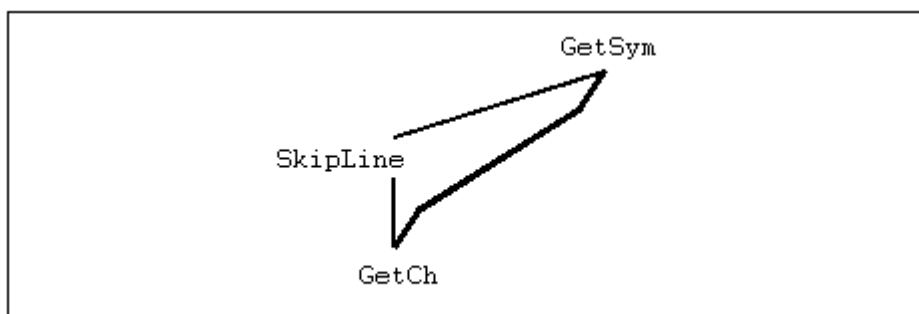
A level skip is illustrated in the call graph by a broken edge: *GetSym* component calls *SkipLine* and *GetCh* components, *SkipLine* also calls *GetCh*.

A level skip corresponds to a call between components separated by hierarchical levels (here there is an intermediate level - *Skipline* between *GetSym* and *Getch*).

In Logiscope **Viewer**, the level skips are displayed in blue.

– direct recursive call
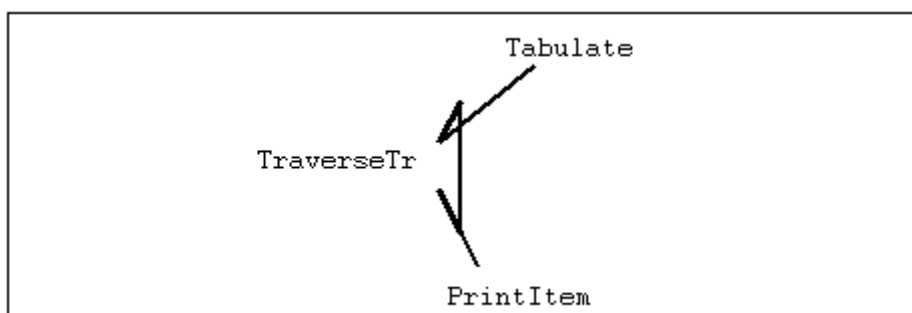
The *TraverseTr* component calls itself. It is recursive or re-entrant (depending on the language).

In Logiscope **Viewer**, the recursive calling relations are displayed in red.

– indirect recursive call
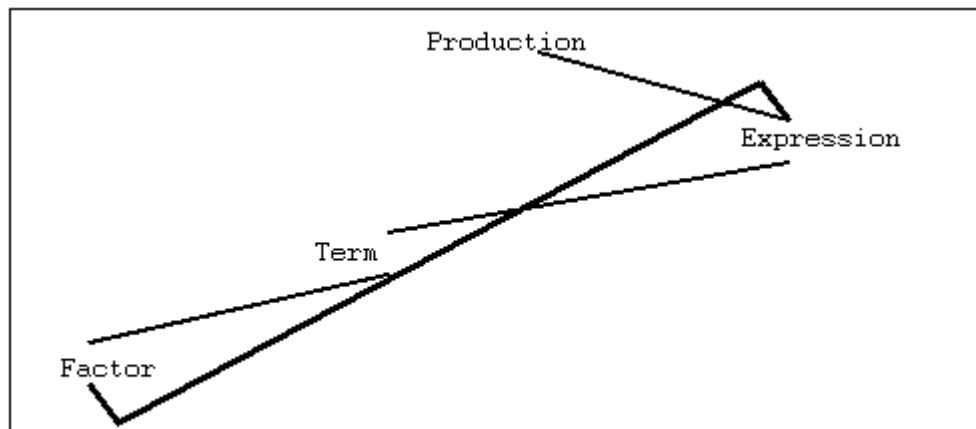
The Expression component calls Term which calls the Factor component. The Factor component calls Expression. Expression is also considered as a recursive or re-entrant component but in an indirect way.

Several operations are possible on a call graph, for example:

– **component numbering**,

– **view of the graph from a particular component**,

– **calling/called component list**,

– **reduction,**

– **simplification by node grouping,**

These functions are described in the following pages.

2.2   Component Numbering

Component identification numbers given in the component list window are linked to call graph nodes.
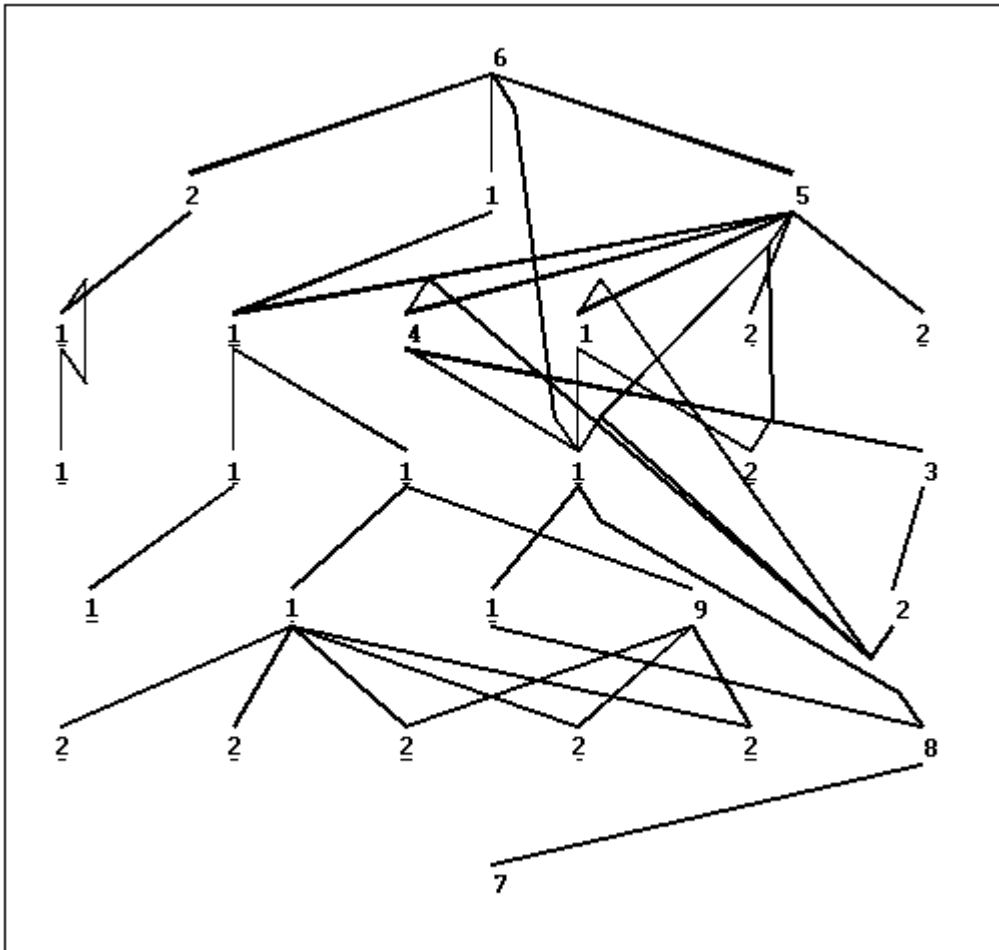
Numbering Call Graph Nodes for Figure 1

2.3   Relative Call Graph

By default, the view of the graph is displayed from the virtual root considered as calling the main root(s).

Any component can be selected as the new root from which the following graph can be displayed:

• the graph of components called by the root and calling the root and their call relations.
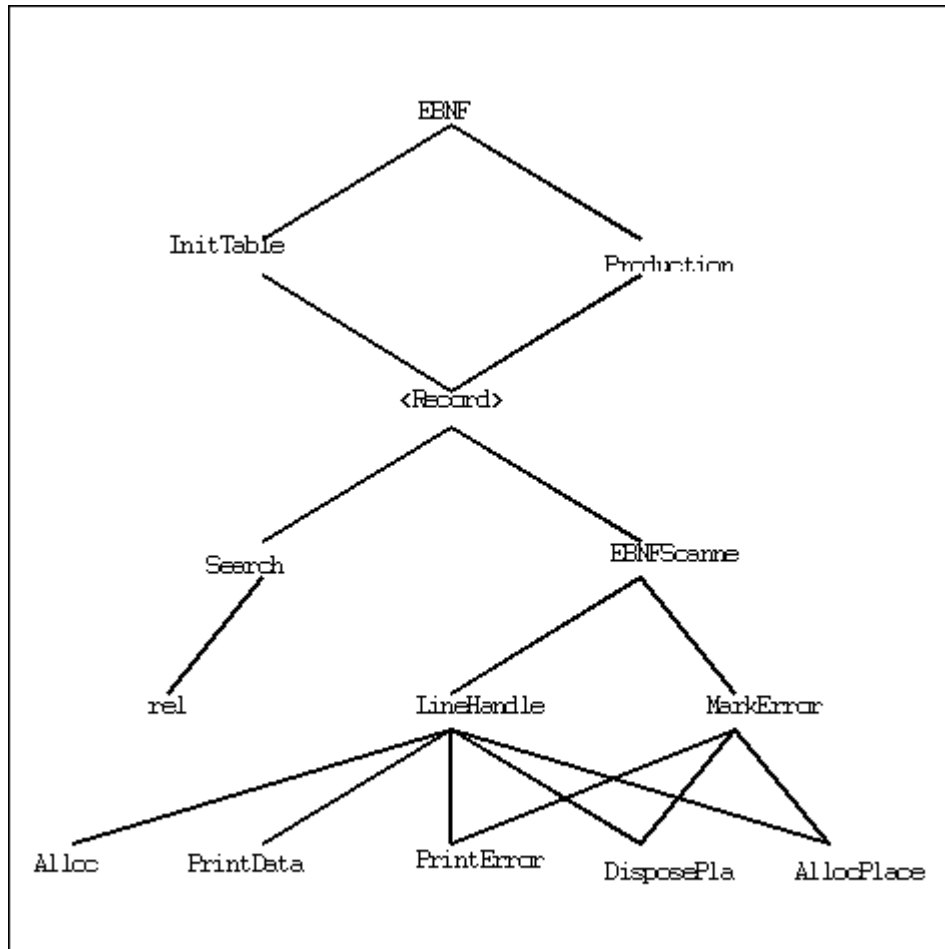
Graph of Components Calling and Called by the <Record> Root

2.4   Calling/called Component List

This function lists the roots of the application call graph, this is useful to know the names of the roots in complex applications with graphs which are difficult to read on the screen. Only one level of calls is displayed.

2.5   Removal of External Components

This operation removes non-analyzed components from the call graph so that only the calling relationships between selected components are displayed.
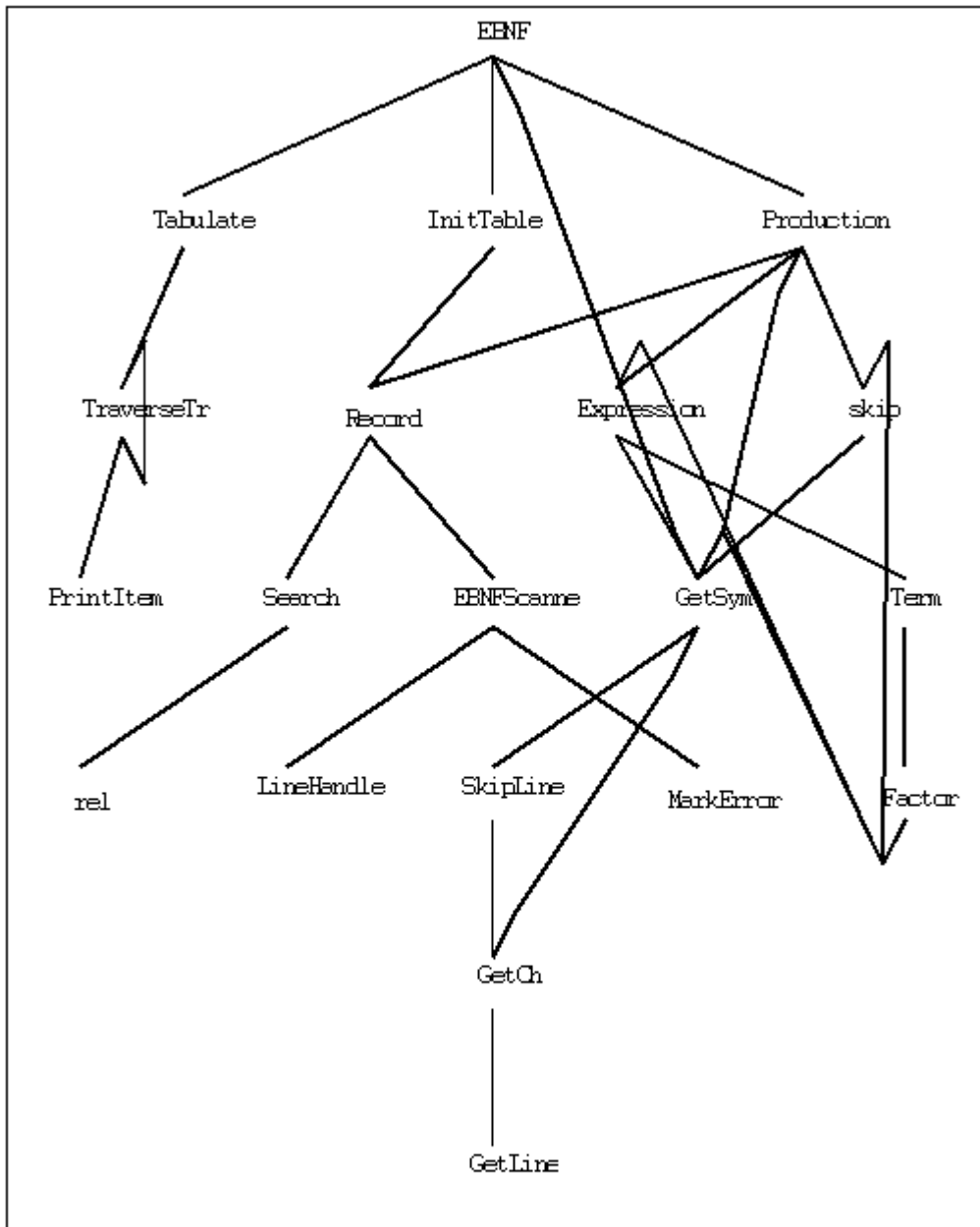
Reduced Call Graph

2.6 Node Grouping

This operation groups together, in a single node, all components selected either

• according to a common part or all of their names,

• according to the files they belong to, or

• according to their complexity properties.

In this way, components from the same file, the same package or the same procedure, depending on the programming language used, can be grouped together.

The name of the component group is set between square brackets. The component group name is user-defined. The default component group identifier is a number.
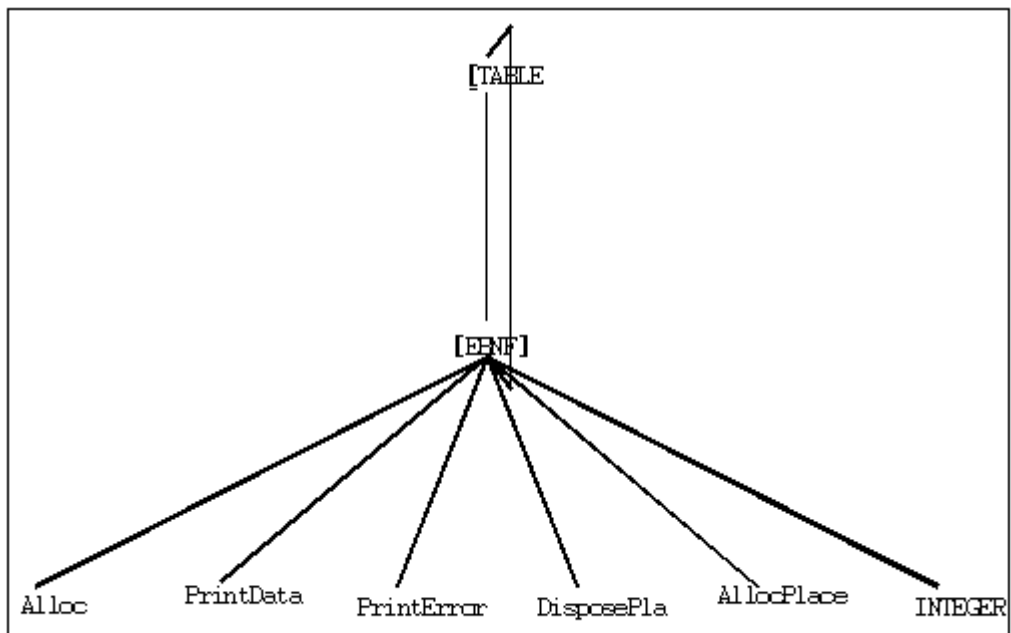
Simplified Call Graph with two node groups: [TABLE] groups all components beginning

with the Table character string and [EBNF] and all components beginning with EBNF

Chapter 3

The Control Graph

3.1 Introduction

Many international standards highly recommend the use of graphic abstraction to evaluate the quality of software product.

For instance, the IEC international standard for *Functional safety of electrical / electronic / programmable electronic safety related systems* recommend control flow analysis to "*detect poor and potentially incorrect program structures*" [61508-7].

According to [61508-7] section C.5.9, the description of this testing technique is the following:

"Control flow analysis is a static testing technique for finding suspect areas of code that do not follow good programming practice. The program is analysed producing a directed graph which can be further analysed for

– inaccessible code, for instance unconditional jumps which leaves blocks of code unreachable;
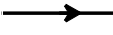
– knotted code. Well-structured code has a control graph which is reducible by successive graph reductions to a single node. In contrast, poorly structured code can only be reduced to a knot composed of several nodes."

Logiscope *QualityChecker* fully supports this static analysis testing technique by providing the control graphs of each function or method defined in the project under analysis.

3.2 Definitions

A control graph displays the logical structure of a software component: function, method or procedure. It consists of nodes, representing statements, and edges representing the transfer of the control flow between nodes.

Nodes are represented differently according to the semantics they convey. The meaning of graphic symbols is as follows:

☐	a sequence of *n* sequential statements.
⊠	main or auxiliary entry or exit of a component.
◇	a test (generally beginning of a control structure) with a # inside for preprocessor tests.
△	end of a control structure with a # inside for preprocessor control structures.
	beginning of an exception sequence (Ada).
	end of an exception sequence (Ada).
	a real-time clause *(accept* statement in Ada).
	a call to an Ada *entry*.
	raising of an exception (*raise* statement in Ada).
▨	a break in a sequence (branch).
△˥	a break in a sequence (unconditional branching).
↓	symbol added to a node referencing a call
	An edge is represented by: ⟶
▨	reduction of a structure on a control graph.

Control structures of analyzed code are represented by combining nodes and edges. Combinations depend on structures expressed by programming languages. For example, the *switch* control structure will not be found in Ada control graphs.
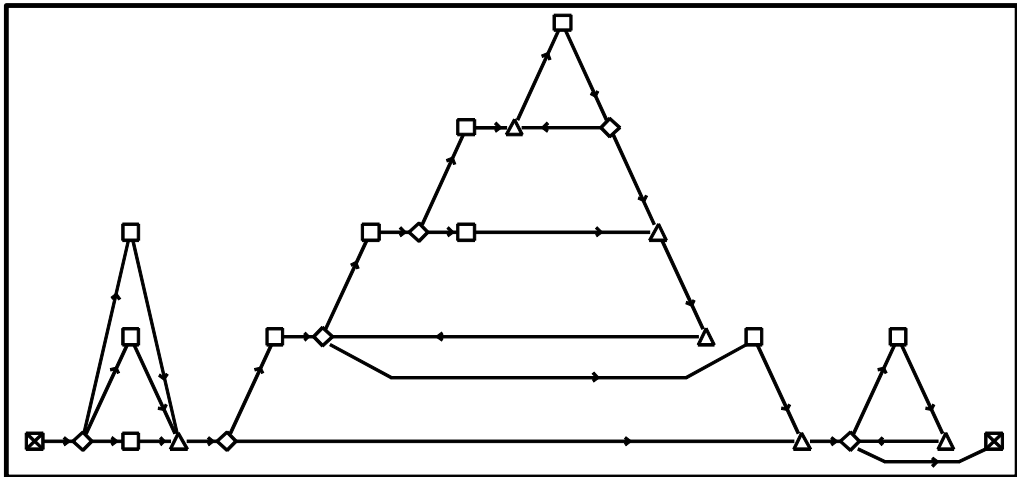
Fig.1 Example of a control graph

Main structures handled by the Logiscope control graph are:

while - do - end of

repeat until

infinite

ITERATIVE STRUCTURES

if then - end if

if then - else - end if

case of - end of case

switch (in C)

if then - elsif - end if

SELECTIVE STRUCTURES

branch

conditional branch

main entry / auxiliary entry

auxiliary exit / main exit

SPECIFIC STRUCTURES

exception processing

Ada raise

real-time clause

SPECIFIC STRUCTURES
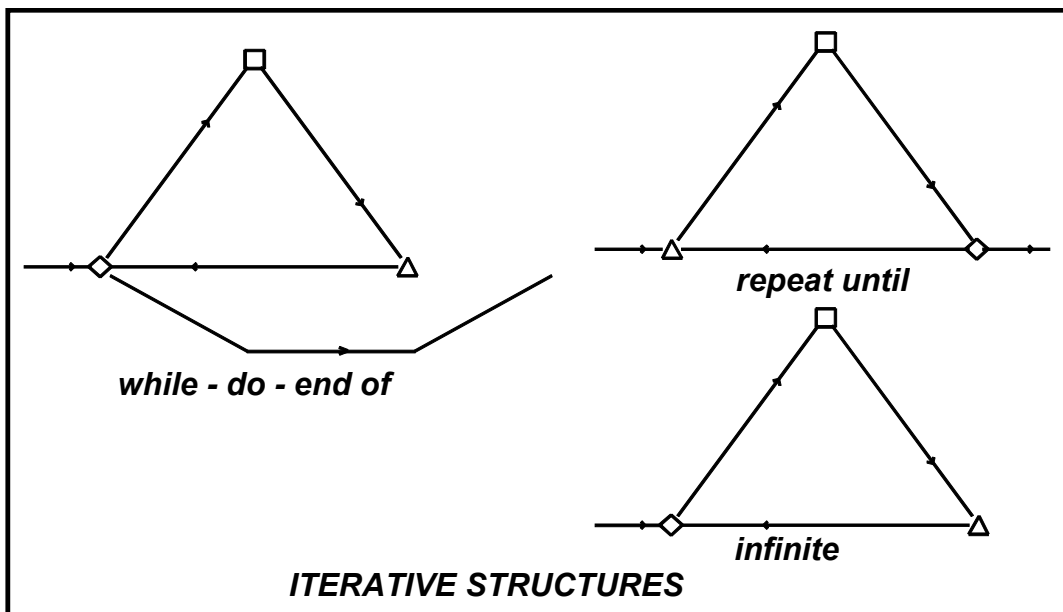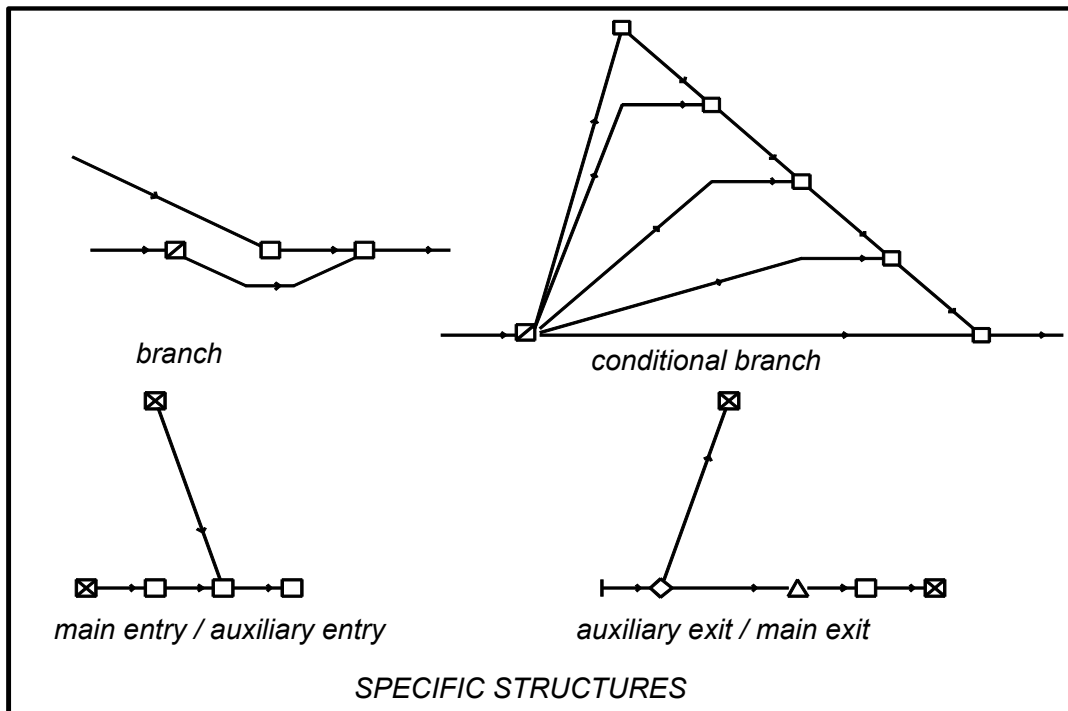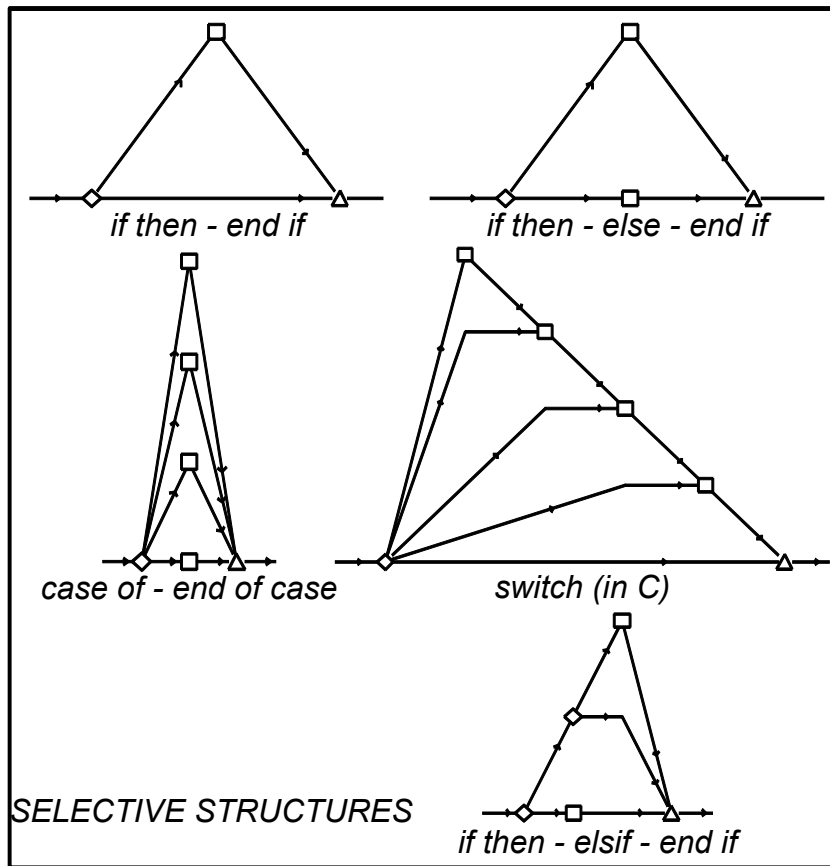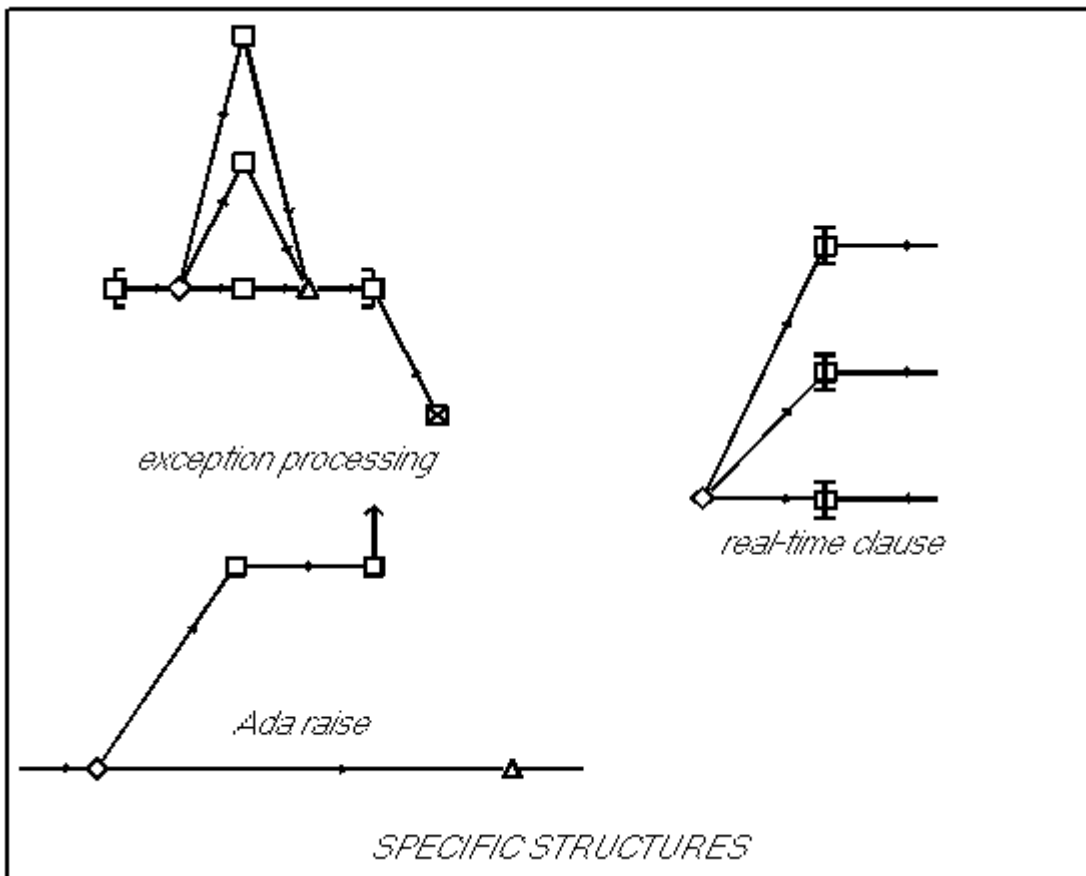
Specific structures from control graphs presented in the last two figures are language-specific representations and do not correspond to typical algorithm structures.

Although control graphs are produced by programs written in different languages, they may look identical.

The control graph is a representation of the logic of the component at a given time. The representation is not language-dependent.
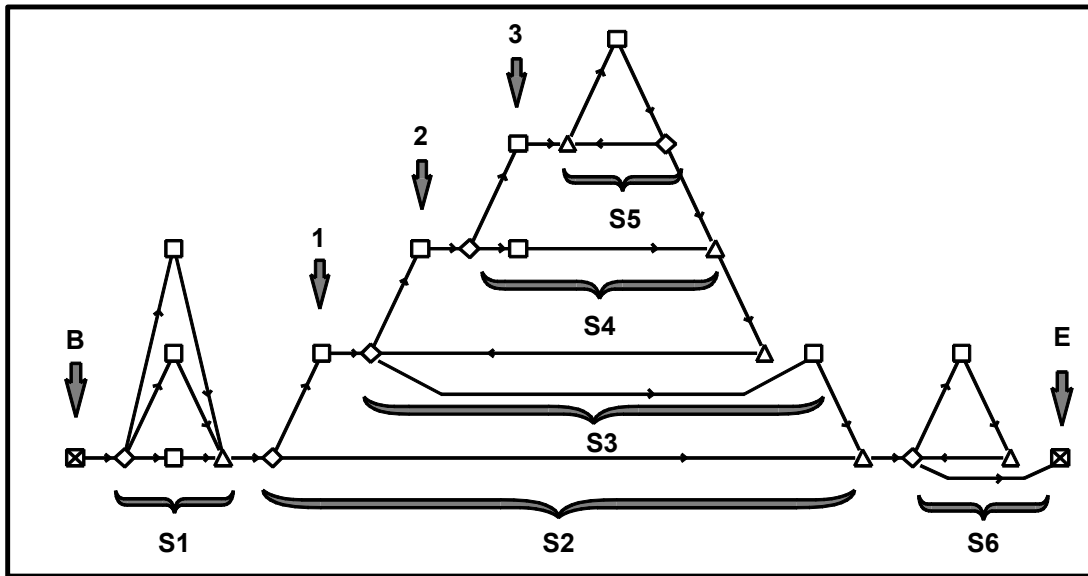
Interpreting a control graph

The control graph above should be interpreted as follows:

– **B**: beginning of the component,

– **S1**: *Switch - End of Switch* type structure

– **S2**: *If-Then-End If* structure comprising:

    1: block of sequential statements

    S3: *While-Do-End of While* structure with statements

    2: block of sequential statements

    S4: *If-Then-Else-End If* structure comprising

    3: block of sequential statements

    S5: *Repeat Until* type structure

– **S6**: a *While-Do-End of While* structure (**E**)

– **E**: End of the component.

Various requests can be performed on a control graph:

– **textual representation**,

– **node numbering**:

    – with statement numbers,

    – with source code line numbers,

– **structured view,**

– **graph reduction**,

– **intrinsic characteristics**,

– **zoom**.

3.3   Pseudo Code

This is a textual representation of the displayed graph. The pseudo code terms (*if then...*) can be modified from the Quality Model file. This makes it possible to obtain an interpretation that is close to the source program or to an algorithmic view, etc. The textual representation of the control graph in Fig. 1 is given below:

```
Begin
   Case
      When (Condition) ==>
         2 Statement (s);
      When (Condition) ==>
         2 Statement (s);
      When (Condition) ==>
         2 Statement (s);
   End of Case;
   If (Condition) Then
      1 Statement (s);
         For (Condition) Do
            1 Statement (s);
            If (Condition) Then
               1 Statement (s);
            Else
               1 Statement (s);
               Repeat
                 1 Statement (s);
               Until (Condition);
            End If;
         End For;
         1 Statement (s);
      End If;
      While (Condition) do
         1 Statement (s);
      End of While;
End;
```

When viewing this textual representation, it is possible to select a pseudo code statement; the corresponding node is highlighted in the control graph. Selecting the control graph node also highlights the pseudo code line.

3.3.1 Instruction Numbers

Numbers appearing on graph nodes are also displayed in the corresponding pseudo code. Note that this numbering takes into account the number of statements in the blocks.
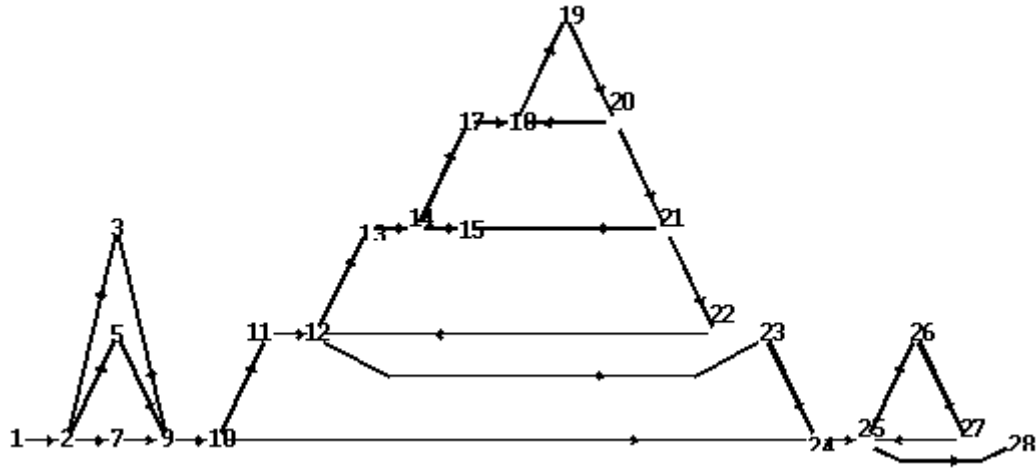

```
 Begin
2   Case
        When (Condition) ==>
3          2 Statement (s);
        When (Condition) ==>
5          2 Statement (s);
        When (Condition) ==>
7          2 Statement (s);
9   End of Case;
10 If (Condition) Then
11     1 Statement (s);
12     For (Condition) Do
13         1 Statement (s);
14         If (Condition) Then
15             1 Statement (s);
           Else
17             1 Statement (s);
18             Repeat
19                1 Statement (s);
20             Until (Condition);
21         End If;
22      End For;
23        1 Statement (s);
24     End If;
25     While (Condition) do
26         1 Statement (s);
27     End of While;
28 End;
```

Graph and pseudo code with statement numbers

3.3.2 Line Numbers
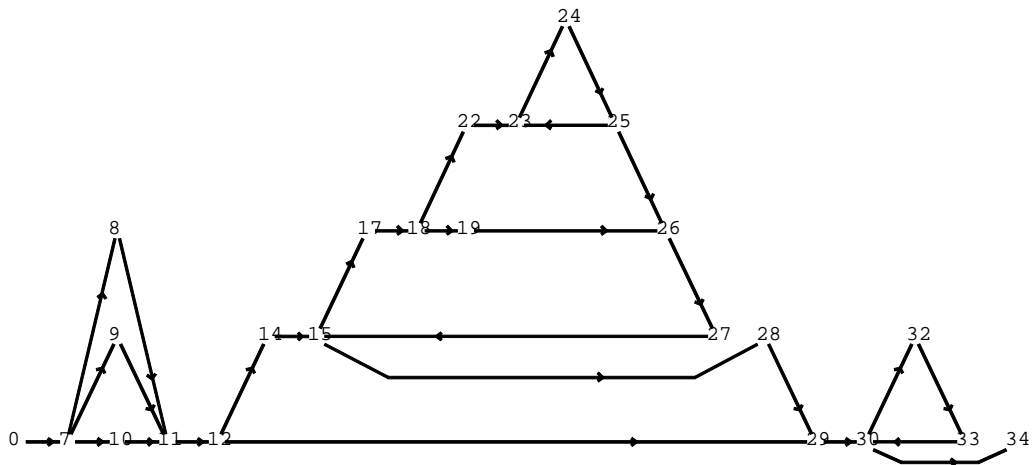

```
   Begin
7  Case
       When (Condition) ==>
8         2 Statement (s);
       When (Condition) ==>
9         2 Statement (s);
       When (Condition) ==>
10         2 Statement (s);
11 End of Case;
12 If (Condition) Then
14     1 Statement (s);
15     For (Condition) Do
17        1 Statement (s);
18        If (Condition) Then
19           1 Statement (s);
           Else
22           1 Statement (s);
23           Repeat
24              1 Statement (s);
25           Until (Condition);
26        End If;
27     End For;
28      1 Statement (s);
29    End If;
30    While (Condition) do
32      1 Statement (s);
33    End of While;
34 End;
```

Graph and pseudo code with source line numbers

3.4   Structured graph

Any algorithm can be expressed in the form of a succession of sequences, iterations and selections.

Due to the fact that certain programming languages do not allow direct translation of algorithmic control structures, the programmer must create his own structures by means of structures such as *If Then - End If* and branches *(GoTo)*.

Since the early days of computing, programming rules have undergone considerable changes: program structuring is now of the utmost importance and code readability is a major objective. But the notion of structured programming is relatively recent and certain applications do not obey these new rules.

Certain applications, for reasons of performance, are developed according to specific programming rules advocating the use of simple but more efficient structures rather than advanced control structures.

Control graphs of such applications may be viewed as critical: their structure does exist but is expressed in a different way.

The structured view of control graphs makes possible to find underlying structures and to show structuring that does not appear explicitly. For example, it can reveal which *GoTo* statements are indicative of structured programming and which *GoTo* are not.

3.4.1 Restructuring Patterns

Several patterns are possible:

– combinations of *If - Then - Else - End - If* and branches expressing selective structures,

– combinations of *If - Then - End - If* and branches representing iterative structures.

```
if   (condition)  then
      go to Label
endif
      [1]
Label: [2]
```


The code and the control graph above express the structured view illustrated below.

```
if  not (condition)  then
      [1]
endif
[2]
```

```
if    (condition)  then
      [1]
      go to Label
endif
[2]
Label: [3]
```

The code and the control graph above express the structured view illustrated below.0


```
if    (condition)  then
      [1]
else
      [2]
endif
      [3]
```

```
if (condition) then
    [1]
    go to Label1
endif
 [2]
 go to Label2
Label1: [3]
Label2: [4]
```


The code and the control graph above express the structured view illustrated below.

```
if (condition) then
    [1]
    [3]
else
    [2]
endif
[4]
```

```
if (condition)  then
     [1]
else
     [2]
     go to Label
endif
     [3]
Label: [4]
```

The code and the control graph above express the structured view illustrated below.


```
if (condition)  then
     [1]
     [3]
else
     [2]
endif
[4]
```

Label1: [1]
  if (condition) *then*
    go to Label1
  endif

[1]

The code and the control graph above express the structured view illustrated below.

Repeat
  [1]
until not (condition)

[1]

```
Label1: if  (condition) then
            go to Label2
        endif
        [1]
        go to Label1
Label2: [2]
```

[1]

[2]

The code and the control graph above express the structured view illustrated below.


```
while not (condition) do
        [1]
end of while
[2]
```

[1]

[2]

```
Label1: [1]
        if (condition) then
            go to Label2
        endif
        go to Label1
Label2: [2]
```

[1]   [2]

The code and the control graph above express the structured view illustrated below.

```
loop
    [1]
    if  (condition) then
            go to Etiq
    endif
    loop end
Label:  [2]
```

[1]

[2]
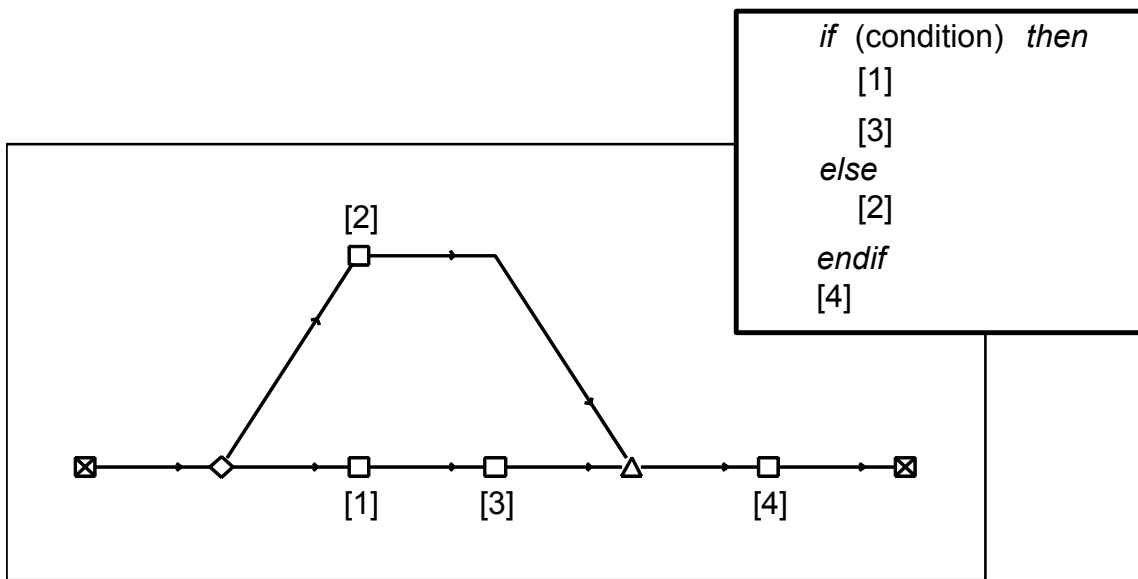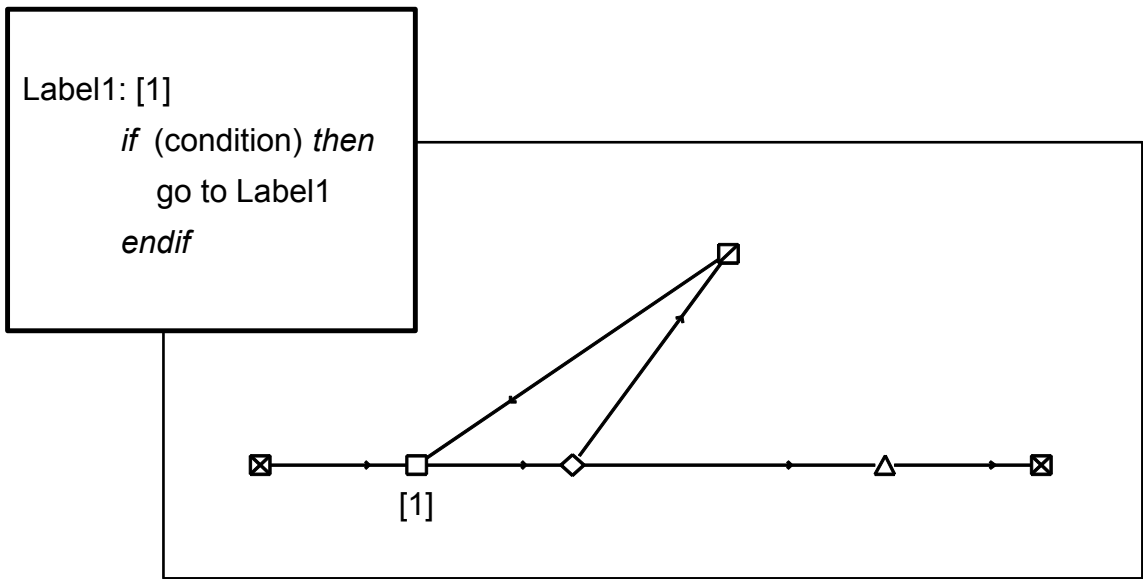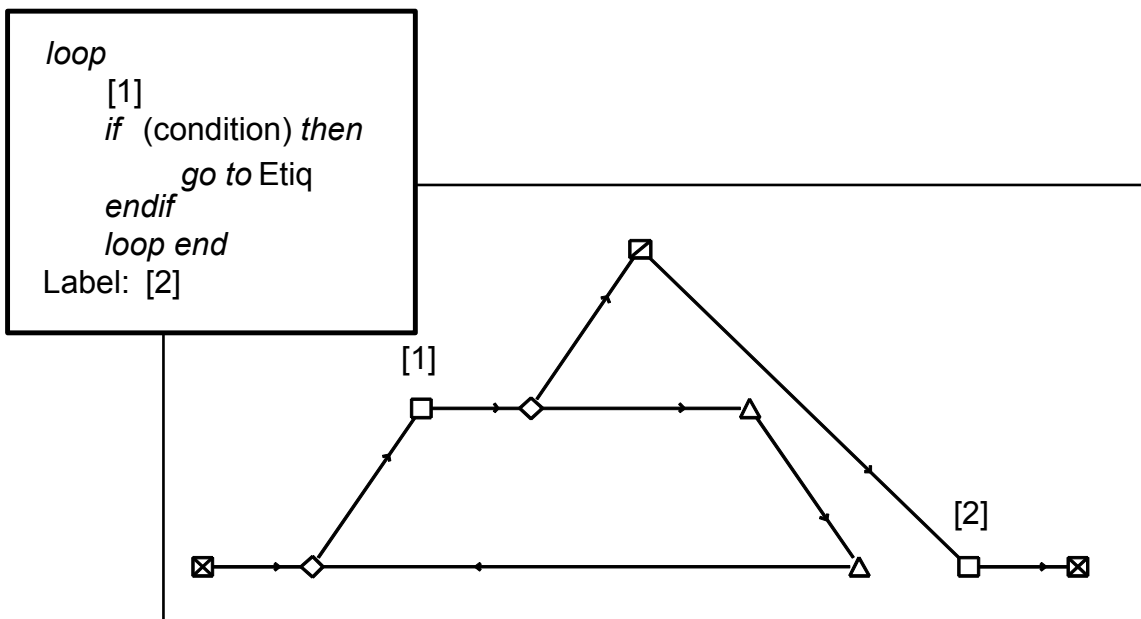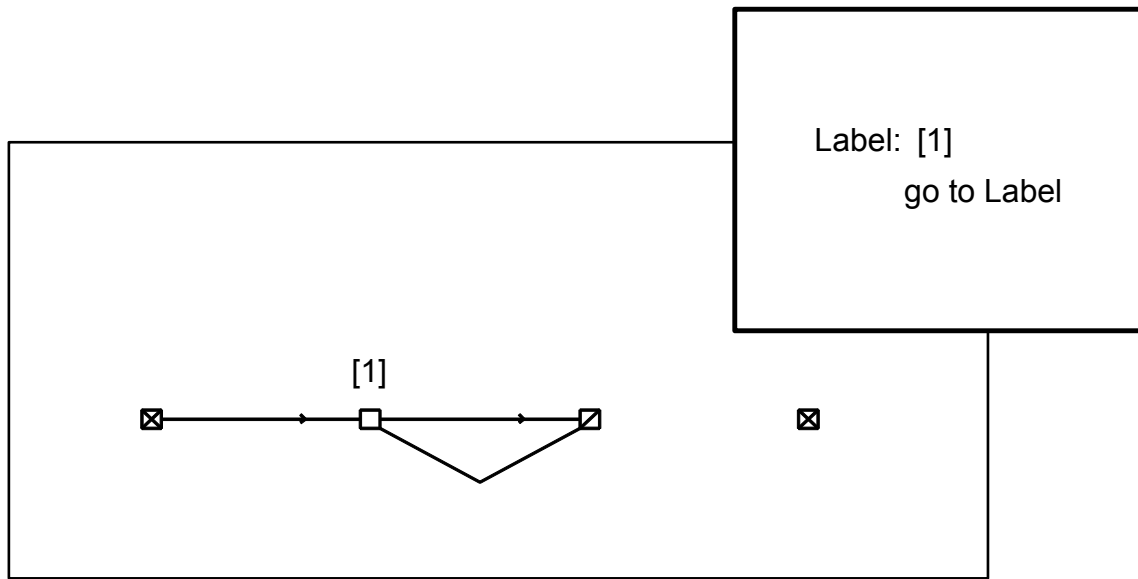
The code and the control graph above express the structured view illustrated below.

Applying these different restructuring patterns to a control graph containing branches (◿) helps to assess the underlying structure (see the following two figures).

Initial control graph

Structured view of the same control graph

3.5　Reduction

The purpose of reducing a graph is in many ways the same as that of obtaining a structured view: to check that a program complies with rules of structured programming.

A control graph which, by successive simplifications, can be reduced to a graph whose cyclomatic number (V(G)) is 1, is said to be structured. Otherwise it is not structured.

The cyclomatic number of a graph reduced to the maximum has an **essential complexity** *equal to 1, indicating that structured programming rules have been applied.*

If the control graph is not structured, reduction will clearly show elements that do not comply with the rules.

3.5.1　Principle

The principle of control graph reduction is to simplify its most deeply nested structured subgraphs (one entry point and one exit point) into a single node.

Four cases result in an non-structured control graph [McCABE 76] are:

– branching into a selective structure,

– branching out of a selective structure,

– branching into an iterative structure,

– branching out of an iterative structure.

Let us look at the following control graph:
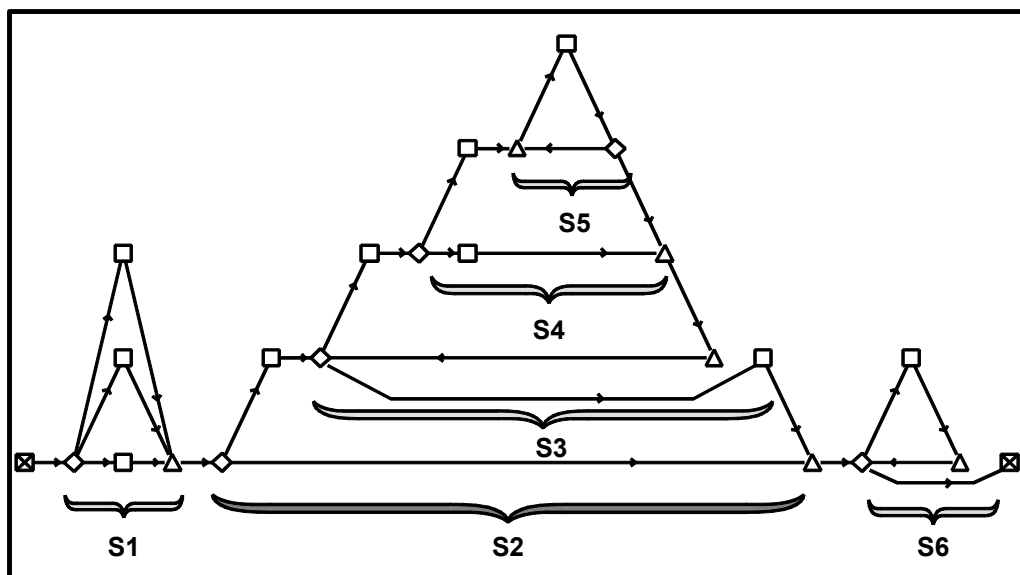
Control Graph Before Reduction

*– first step: the structures without nesting and that complies with the rules of structured programming (**S1** and **S6**) and the most deeply nested structure (**S5**) are reduced. These reductions are symbolized on the new graph by the following hatched squares ◪ (See the following figures).*
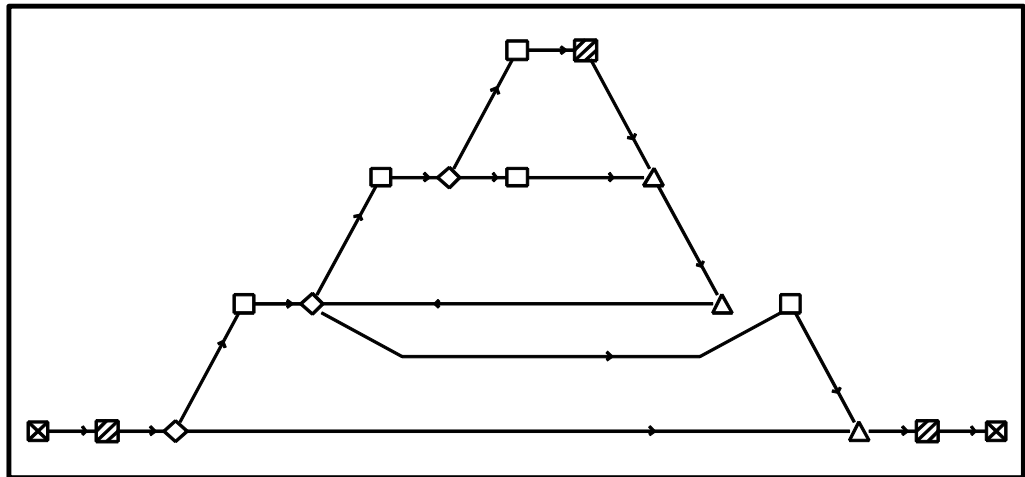
Control Graph after the First Reduction

*– from the second to the last step (see the following three figures): the most deeply nested structure (**S4** then **S3**) is reduced.*
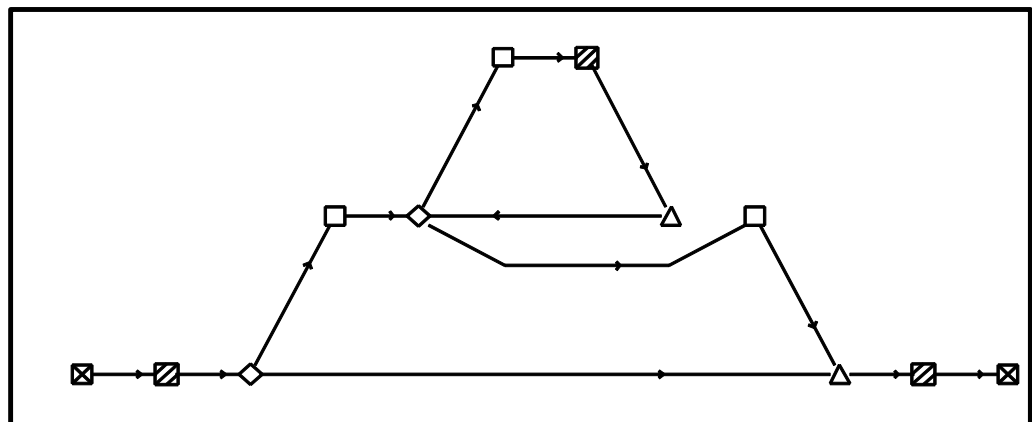
Control Graph after the Second Reduction
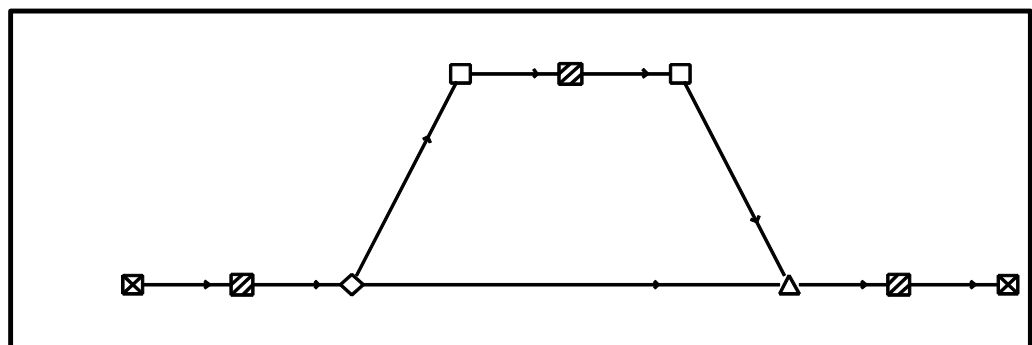
Control Graph after the Third Reduction

Control Graph after Complete Reduction

Reduction can also be performed on the structured view of a control graph. But care must be taken: even if it is possible to simplify the structured view to the maximum, this does not mean that the control graph can always be reduced (see the following two figures).

Example

Fig. 2 Control Graph

Control Graph after Complete Reduction

It is not possible to reduce further this control graph.

Now let us look at the reduction of the structured view of the same graph.

Structured View of the Control Graph Shown on Fig. 2
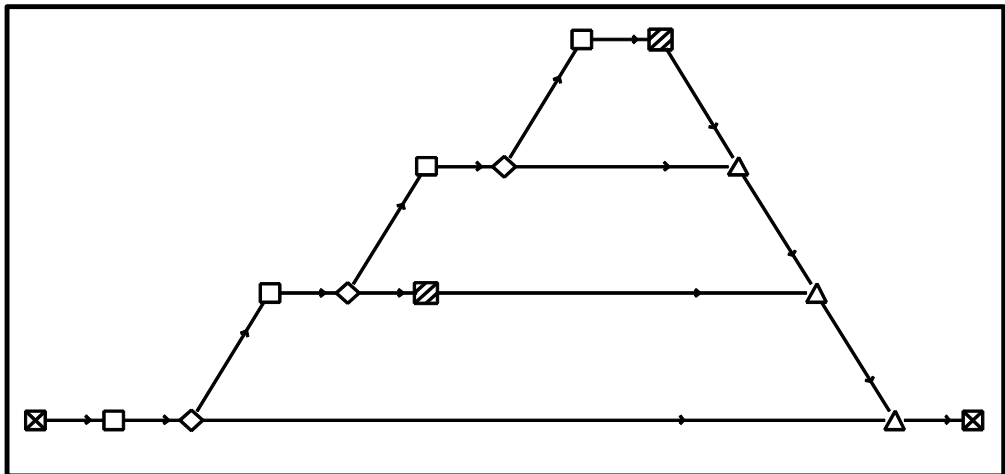
First Reduction
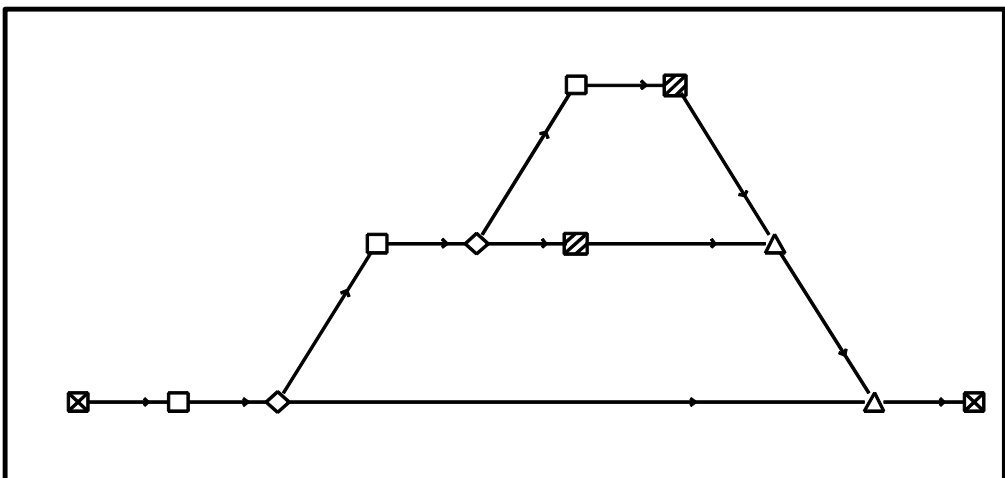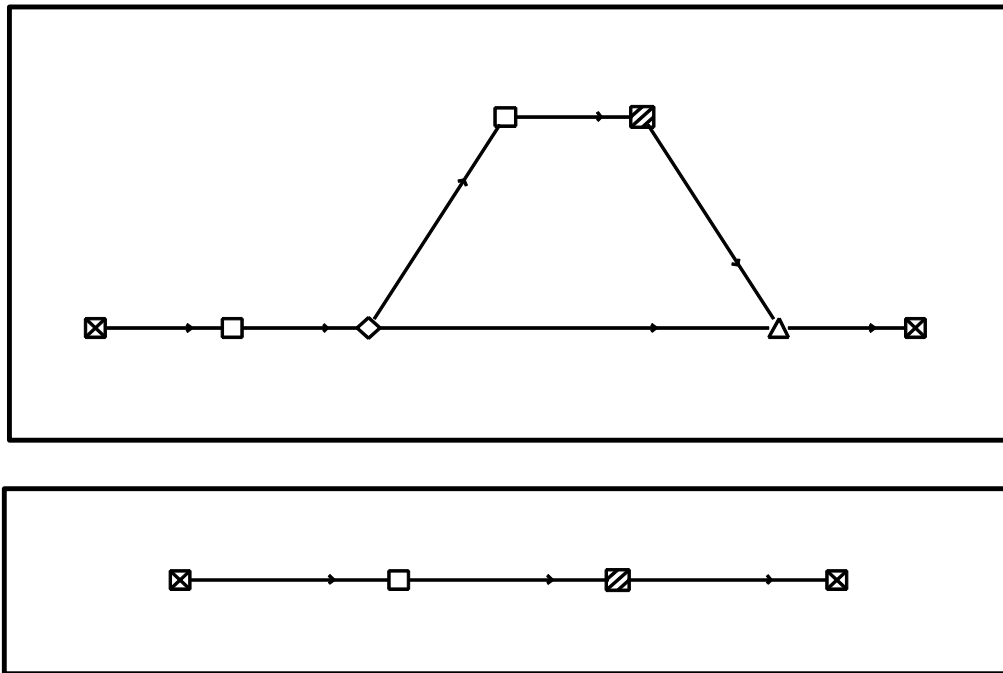
Complete Reduction is possible

The structured view of the graph in Fig. 2 respects programming rules of structure.

3.6 Intrinsic Characteristics

These are characteristic values of the current control graph (original control graph, structured or reduced view).

These measurements are:

– all measurements that can be performed on a control graph,

– the number of reduced nodes,

– the number of restructuring performed, distributed by type of structure obtained.

Only measurements defined for the current graph with a non-null value will be displayed.

Chapter 4

Evaluating Quality Using Source Code Metrics

4.1   Introduction

Many international standards highly recommend the use of source code metrics to evaluate the quality of software product.

For instance, the IEC international standard for *Functional safety of electrical / electronic / programmable electronic safety related systems* recommends source code complexity metrics to *"predict the attributes of program from properties of the software itself or from its development or test history"* [61508-7].

According to [61508-7] section C.5.14, the description of this testing technique is the following:

"These models evaluate some structural properties of the software and relate this to a desired attribute such as reliability or complexity. Software tools are required to evaluate most of the measures. Some of the metrics which can be applied are given below:

> *– graph theoretic complexity – this measure can be applied early in the lifecycle to assess trade-offs, and is based on the complexity of the program control graph, represented by its cyclomatic number;*

> *– number of ways to activate a certain software module (accessibility) – the more a software module can be accessed, the more likely it is to be debugged;*

> *– Halstead type metrics science – this measure computes the program length by counting the number of operators and operands; it provides a measure of complexity and size that forms a baseline for comparison when estimating future development resources;*

> *– number of entries and exits per software module – minimising the number of entry/exit points is a key feature of structured design and programming techniques."*

Logiscope *QualityChecker* fully supports this static analysis testing technique by providing source code metrics as well as quality modeling to allow rating of software components.

4.2 Modeling Quality

In order to evaluate software quality, it must first of all be modeled. The modeling approach used by Logiscope is similar to those defined by Boehm [BOEHM, 75] and McCall [MCCALL, 77] and compliant with the ISO/IEC 9126-1 international standard [91-26].

According to this approach, software quality can be defined as a set of characteristics which:

- are important to the user: quality factors,
- can be decided by the designer: quality criteria,
- can be measured for verification purposes: quality metrics.

The advantage of this approach is that quality is:

- specified in terms of factors,
- designed in terms of criteria,
- built with the help of programming rules,
- assessed by means of metrics.

For example, if maintainability is among the most crucial quality criteria of an application, our first aim will be to find:

> (a better) *possibility of detecting, locating and correcting anomalies, of introducing minor changes and then of accomplishing the required functions with the anticipated resources [GAM - T17 (V2) July 1989].*

In this case, the most crucial quality characteristics are determined from the maintenance engineer's point of view which will define the *quality factor.*

In order to satisfy this factor, software designers will opt for, among other things, a self-descriptive application defined as follows:

> (...) *attributes of the software that provide explanation of the implementation of a function [Ref. MCCALL, 77].*

Quality characteristics determined by software designers are *quality criteria.*

In order to reach a sufficient level of self-documentation, the following programming rule will be respected when building the application:

the source code must contain at least 1 comment for 5 executable statements and 1 comment at the most for 1 statement.

In order to check if this rule is respected throughout development, Logiscope will measure the frequency of comments (number of comments/number of statements). This frequency must be between 0.2 and 1.

This quality characteristic can be verified: it is a *quality metric*.

A quality factor can be assessed by means of a set of quality criteria.

For example, the self-description, modularity, readability, simplicity criteria, contribute towards satisfying the maintainability quality factor.

In the same way, quality criteria can be assessed by means of a set of quality metrics.

For example, a correct comment frequency value contributes towards meeting the self-describing quality criterion.

These contribution relationships allow the quality of an application to be modeled in tree form:
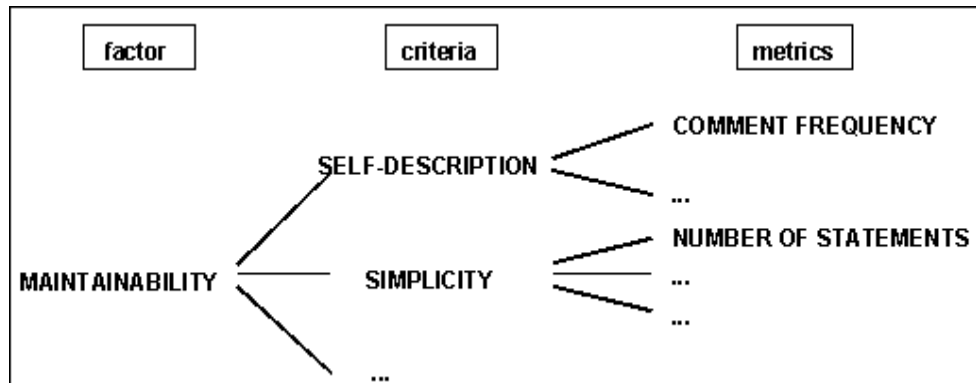
4.3   Quality Evaluation Using QualityChecker

The user can define a quality model adapted to his needs in Quality Model files. Logiscope refers to this file to obtain a specific model to evaluate quality.

On the basis of the defined model, the approach to evaluating quality using Logiscope is broken down into three successive stages:

– **Metrics:** comparison of measured values with limit values previously defined in the Quality Model file on the basis of the programming rules,

– **Criteria**: classification in different categories according to results obtained for metrics related to each criterion in the Quality Model file,

– **Factor**: classification in different categories according to results obtained at criteria level for the factor in the Quality Model file.

4.4 Metrics

Logiscope *QualityChecker* proposes a set of basic metrics that allows to measure the complexity at component or architecture level.

By combining Logiscope metrics, the user can add new metrics to the list of Logiscope metrics.

For example, the "comments frequency" metric can be defined by combining two Logiscope metrics: "number of comments" and "number of statements" as follows: : `com_freq = lc_comm /lc_stat`.

For each metric in the quality model, the user associates limit values indicating minimum and maximum values accepted for the metric.

At this stage of quality evaluation, the Logiscope *Viewer* provides the following results:

– **Kiviat graph**,

– **metrics table**.

4.4.1 Kiviat Analysis
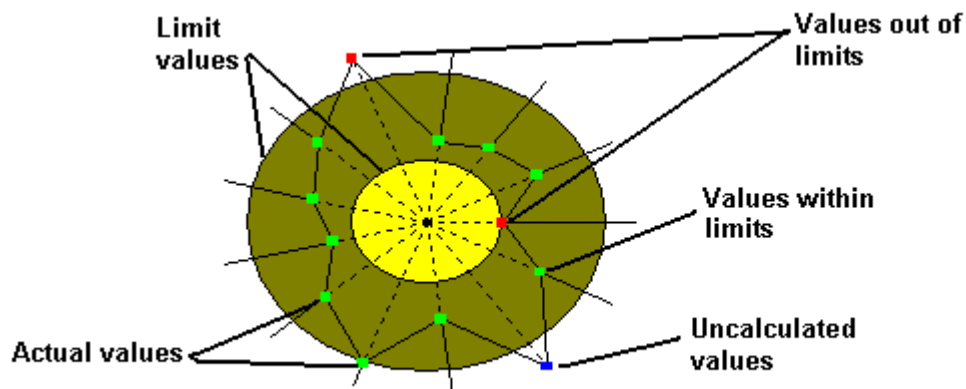
Kiviat Graph

The Kiviat analysis provides a graphic display of the state of an object (component or application) with respect to limit values:

- each axis represents a metric,
- limits are indicated by two circles: the inner circle corresponds to the minimum value accepted, and the outer circle corresponds to the maximum value accepted,
- the polygon links all values obtained for the object analyzed,
- limit values defined and values found for the various metrics are given in the upper left hand corner of the graph (See the graph below),
- the overall assessment of results is immediate. If values are acceptable, the polygon will be drawn between the two circles. Kiviat graphs can thus be compared to a template.

The two following figures respectively illustrate programming that meets the required standards and programming that does not.
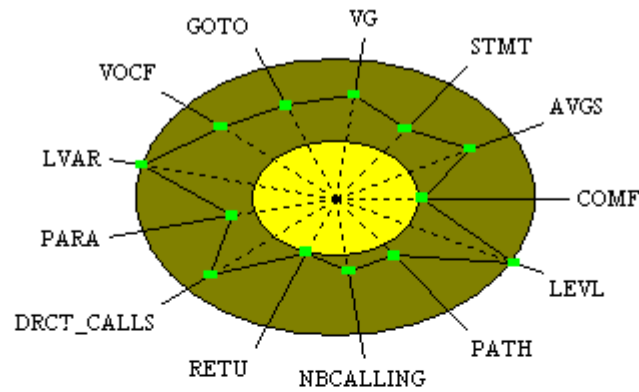
AXIS	LOW.	HIGH.	VALUE
COMF	0,10	5,00	0,18
AVGS	1,00	9,00	5,59
STMT	1,00	50,00	17,00
VG	1,00	10,00	6,00
GOTO	0,00	0,00	0,00
VOCF	1,00	4,00	2,79
LVAR	0,00	5,00	5,00
PARA	0,00	5,00	1,00
DRCT_CALLS	0,00	7,00	5,00
RETU	0,00	1,00	0,00
NBCALLING	0,00	5,00	1,00
PATH	1,00	80,00	13,00
LEVL	0,00	4,00	4,00

Programming meets the standards
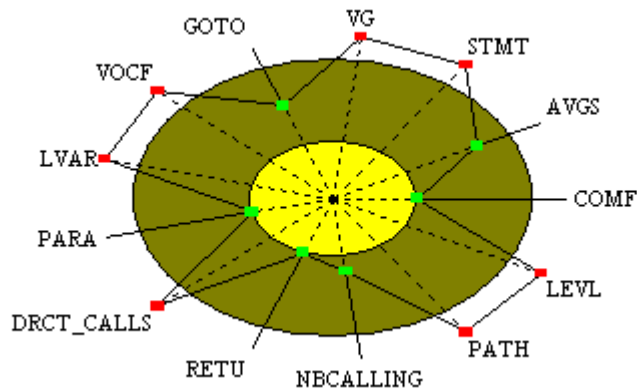
AXIS	LOW.	HIGH.	VALUE
COMF	0,10	5,00	0,12
AVGS	1,00	9,00	6,38
STMT	1,00	50,00	165,00
VG	1,00	10,00	56,00
GOTO	0,00	0,00	0,00
VOCF	1,00	4,00	12,52
LVAR	0,00	5,00	11,00
PARA	0,00	5,00	0,00
DRCT_CALLS	0,00	7,00	10,00
RETU	0,00	1,00	0,00
NBCALLING	0,00	5,00	1,00
PATH	1,00	80,00	32 767,00
LEVL	0,00	4,00	7,00

Programming does not meet the standards

When a metric value cannot be calculated, it is represented by **** in the VALUE *column and does not appear on the graph. Overflow will be represented by a > in the* VALUE *column and the metric is placed on the maximum level on the graph.*

A maximum of 10 characters can be used to display the metrics mnemonics.

4.4.2 Metric Kiviat table

The metrics table provides the same information as the Kiviat graph, but in textual form.

This table contains:

- names of metrics,
- metric mnemonics,
- values measured,
- conformity with limit values: values that are not acceptable are indicated by an asterisk "*" in the last column of the table.

Metrics	Ident.	Value	!
Comments frequency	COMF	0,24	
Average size of statements	AVGS	4,76	
Number of statements	STMT	17,00	
Cyclomatic number (VG)	VG	3,00	
Number of GOTO statements	GOTO	4,00	*
Vocabulary frequency	VOCF	3,00	
Number of local variables	LVAR	2,00	
Number of function parameters	PARA	1,00	
Number of direct calls	DRCT_CALLS	4,00	
Number of RETURN statements	RETU	1,00	
Number of callers	NBCALLING	2,00	
Number of paths	PATH	***	*
Number of levels	LEVL	2,00	

Component Metrics Table

In the Value column, a metric value which cannot be calculated is represented by **** and an overflow by the character >.

4.4.3 Average Kiviat Graph

This Kiviat graph displays average values and standard deviations obtained for metrics for all analyzed components.

At synthesis level, only components with defined values will be taken into account (and not those whose values could not be calculated or those above the capacity level).

The standard deviation each side of the mean value is represented.
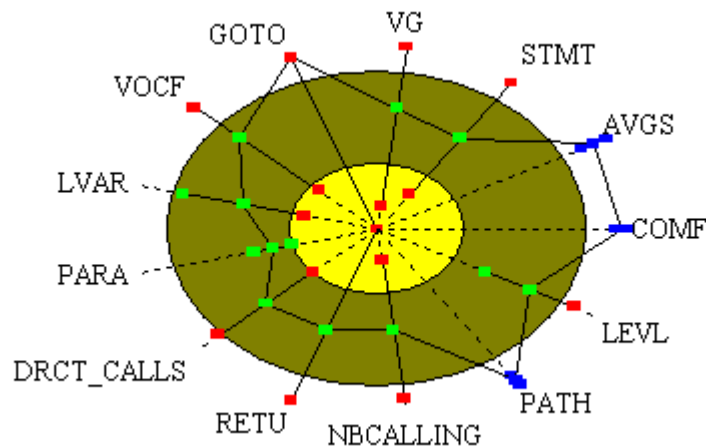
AXIS	LOW.	HIGH.	MIN.	MAX.	AVERAGE
COMF	0,10	5,00	***	***	***
AVGS	1,00	9,00	***	***	***
STMT	1,00	50,00	0,00	165,00	24,65
VG	1,00	10,00	1,00	56,00	6,51
GOTO	0,00	0,00	0,00	29,00	0,92
VOCF	1,00	4,00	1,00	12,52	3,35
LVAR	0,00	5,00	0,00	11,00	2,05
PARA	0,00	5,00	0,00	3,00	0,81
DRCT_CALLS	0,00	7,00	0,00	13,00	3,51
RETU	0,00	1,00	0,00	12,00	0,46
NBCALLING	0,00	5,00	0,00	21,00	1,95
PATH	1,00	80,00	***	***	***
LEVL	0,00	4,00	1,00	7,00	2,78

Kiviat Graph of Averages

Reference values for each metric and the average obtained for all components are indicated below the graph.

Reference values for each metric and the average obtained for all components are indicated in the upper left hand corner.

4.4.4 Average Metrics Table

Metrics table provides the same information as the Kiviat graph of averages, but in textual form. Characteristic values of the analyzed components in this table are:

- average,
- standard deviation,
- minimum value measured,
- maximum value measured.

The percentage of acceptable components and the percentage for which it was not possible to calculate the metric are indicated in the last two columns of the table.

Metrics	Ident.	Average	Std dev.	Min. value	Max. value	% Acc.	% Undef.
Comments frequency	COMF	0,20	0,22	0,00	1,00	65,63	13,51
Average size of statements	AVGS	5,68	1,78	3,44	10,00	93,75	13,51
Number of statements	STMT	24,65	36,09	0,00	165,00	67,57	0,00
Cyclomatic number (VG)	VG	6,51	9,67	1,00	56,00	81,08	0,00
Number of GOTO statements	GOTO	0,92	4,73	0,00	29,00	91,89	0,00
Vocabulary frequency	VOCF	3,35	2,58	1,00	12,52	72,97	0,00
Number of local variables	LVAR	2,05	2,55	0,00	11,00	86,49	0,00
Number of function parameters	PARA	0,81	0,80	0,00	3,00	100,00	0,00
Number of direct calls	DRCT_CALLS	3,51	3,63	0,00	13,00	86,49	0,00
Number of RETURN statements	RETU	0,46	1,97	0,00	12,00	94,59	0,00
Number of callers	NBCALLING	1,95	3,78	0,00	21,00	94,59	0,00
Number of paths	PATH	1 027,97	5 529,23	1,00	32 767,00	88,24	8,11
Number of levels	LEVL	2,78	1,65	1,00	7,00	81,08	0,00

Statistics table

*Non-calculated values will be represented by the characters ******** and overflows by **>**.*

4.4.5 Metrics Distribution

This histogram indicates the distribution of components, according to values of the analyzed metric, between minimum and maximum values measured. The metric limit values are represented by two arrows to show the distribution of components with respect to these limits.
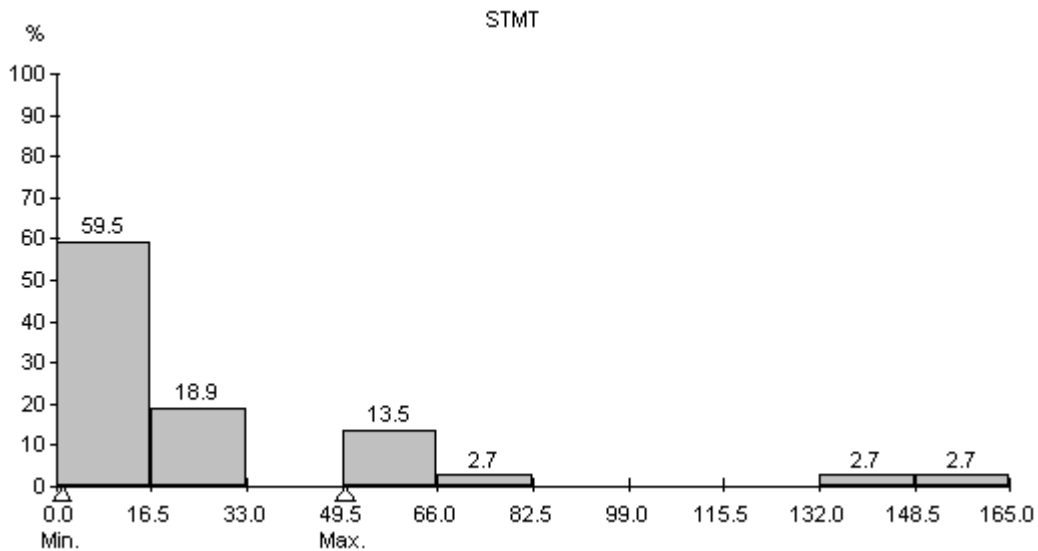
The Logiscope *Viewer* calculates the optimum number of bars, between 1 and 20, according to values measured for the metric. In order to obtain a different view, the user can:

• modify the number of bars,
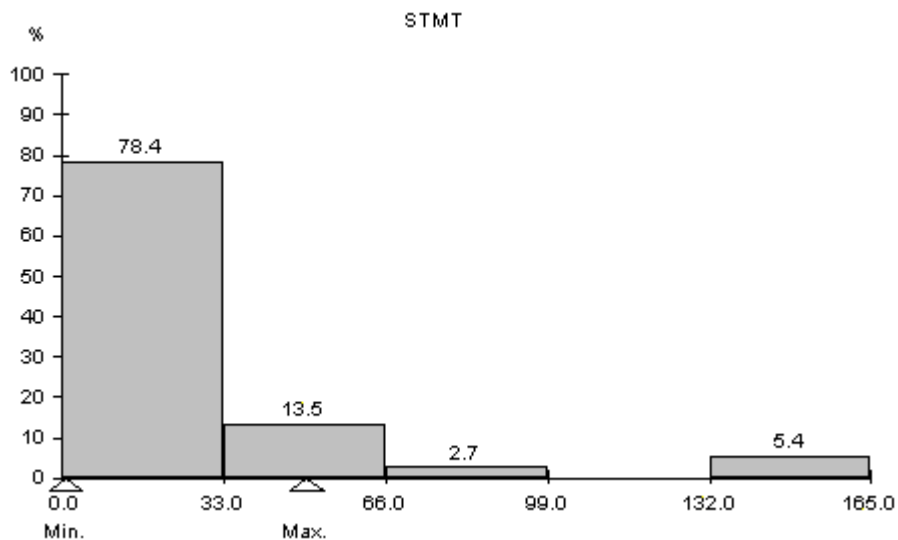
Distribution after modifying the number of classes

- modify the distribution interval for a clearer view of part of the histogram,
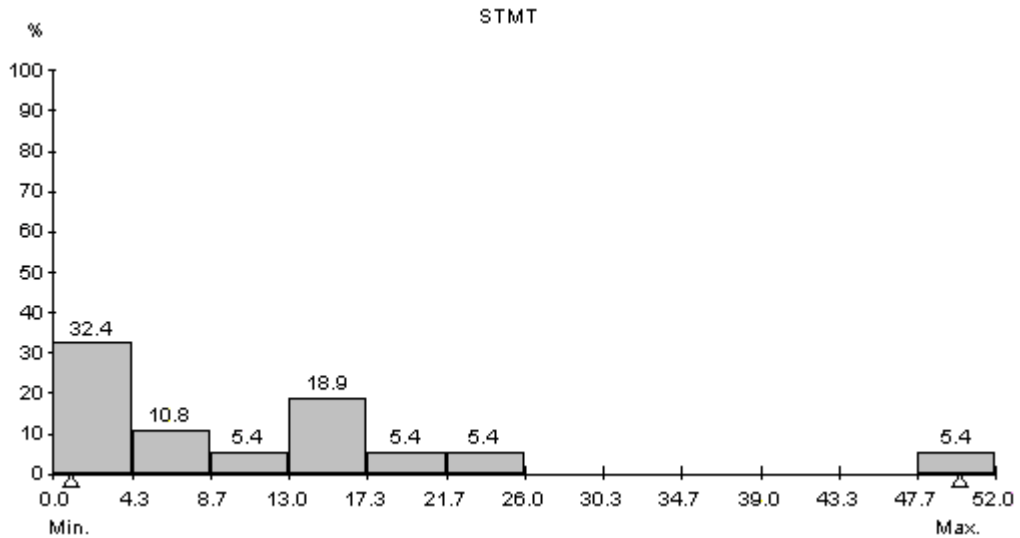
Focus on part of the histogram

- obtain a distribution in three non-uniform classes, the distribution criterion being in conformity with the metric limit values. The three classes are:
  - n [minimum value measured - lower limit[,
  - n [lower limit - upper limit],
  - n]upper limit - maximum value measured].
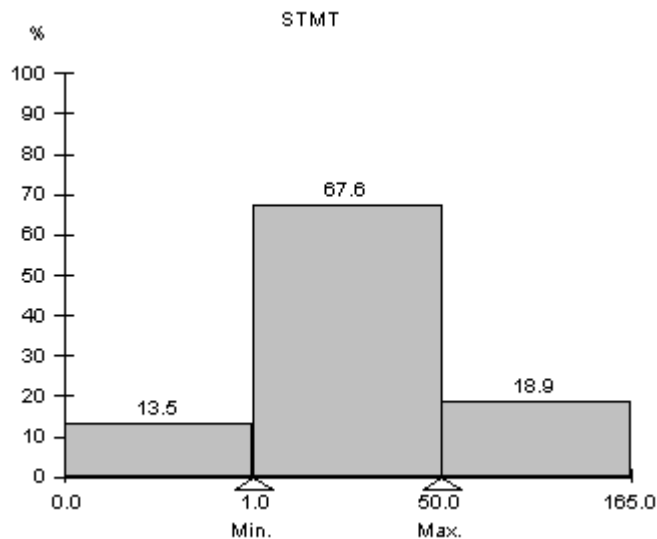
Metrics distribution (limit values)

For each distribution, results are displayed in textual form, as shown below:

VG =>	No.	Name
13,00	15	instruction
15,00	6	consistent
17,00	23	player_plays
21,00	24	player_score
56,00	16	machine_plays

4.5 Criteria

A criterion can be defined by means of a set of Logiscope metrics.

By associating a set of metrics with each criterion in the quality model, the user can associate the "cyclomatic number" and "number of statements" metrics with the simplicity criterion. For example, to create a "simplicity" criterion by combining the ct_vg and lc_stat metrics.

In addition to this, the user can assign a weight coefficient to each metric. The weight coefficient represents the share of the metrics in satisfying the criterion.

For example, if respecting rules established for the cyclomatic number seems more important than respecting rules for other metrics, its metric should be weighted more.

An object is classified in a category according to:

• whether limit values for each metric are respected or not,

• the weight coefficient assigned to each metric.

In this second stage of quality evaluation, Logiscope provides the following results:

• **criteria graph**,

• **criteria distribution**.

4.5.1 Criteria Graph

The criteria graph ranks an object with respect to a set of quality criteria defined in the model. The following information is presented in this graph:

- criterion/ metric associations,
- the metrics position with respect to limit values (see Kiviat graph),
- the category of the object is given for each criterion.
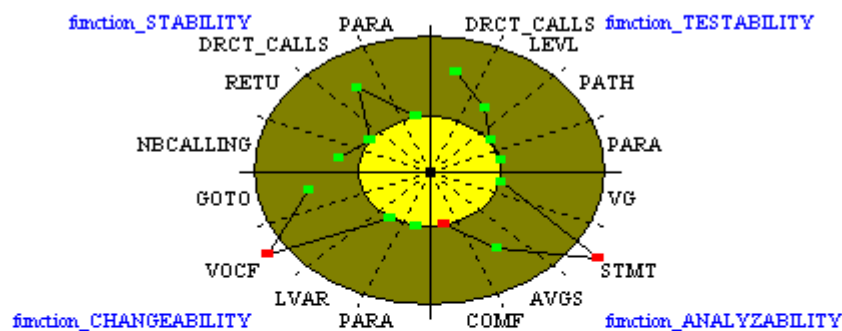
CRITERION	CLASS
function_TESTABILITY	EXCELLENT
function_STABILITY	EXCELLENT
function_CHANGEABILITY	GOOD
function_ANALYZABILITY	FAIR
SYNTHESIS	GOOD

Criteria Graph

- On the component level criteria graph, the category of the object for the quality factor is also given (see below).
- A maximum of 10 characters can be used to display the criteria mnemonics.

4.5.2 Criteria Distribution

This result presents the distribution of all components analyzed with respect to each criterion defined.

In this histogram, each category of the criterion is represented by a column which is proportional to the number of components that belong to the category.
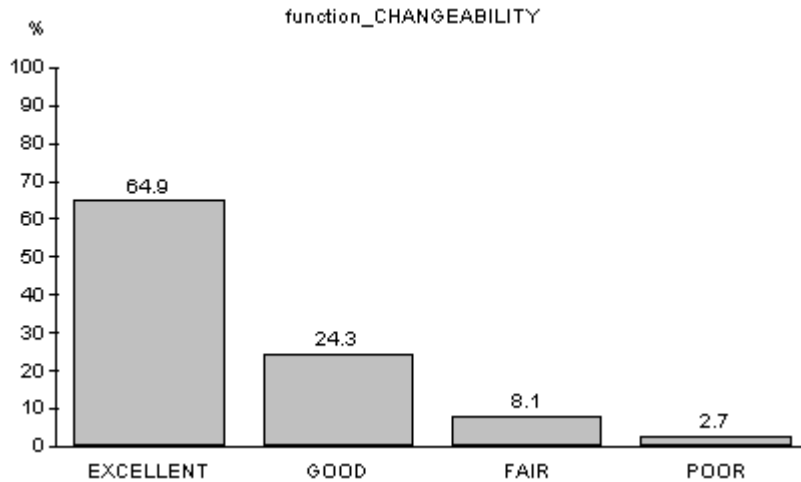
Criteria distribution

For each distribution, results are displayed in textual form, as shown below:

function_CHANGEABILITY =>	No.	Name
EXCELLENT	37	waiting_loop
GOOD	6	consistent

List of components per criteria category (extract)

4.6 Quality Report

The user defines the quality factor as a set of categories that represents the extent to which the factor has been fulfilled. Components are classified according to criteria analysis.

A weight coefficient is associated with each criterion category when it is defined indicating the contribution it makes towards the factor. Thus all defined criteria contribute to the evaluation of the factor (except if the weight coefficient is zero).

The quality report is a pie chart showing the distribution of the components according to the quality factor. It indicates the percentage of components belonging to each category defined.
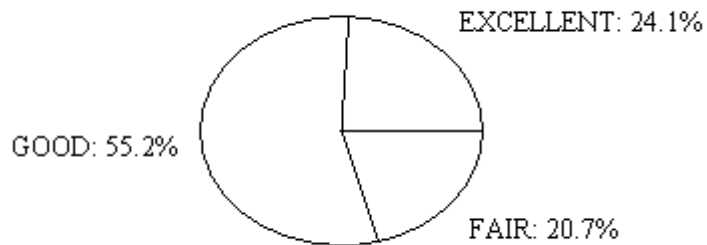
Component Quality Report

The following illustrates the quality report shown in textual form:

function_MAINTAINABILITY	No.	Name =>
FAIR	6	consistent
EXCELLENT	7	end_game
GOOD	8	find_digit

Chapter 5

Standard Metrics Definition

5.1   Introduction

Logiscope *QualityChecker* proposes a set of standard source code metrics. Source code metrics are static measurements (i.e. obtained without executing the program) to be used to assess attributes (e.g. complexity, self-descriptiveness) or characteristics (e.g. Maintainability, Reliability) of the software components (e.g. functions, classes, modules, package, application) under evaluation.

The metrics can be combined to define new metrics more closely adapted to the quality evaluation of the source code. For example, the "Comments Frequency" metric, well suited to evaluate quality criteria such as Self-descriptiveness or Analyzability, can be defined by combining two standard metrics: "Number of comments" and "Number of statements".

The user can associate threshold values with each of the quality model metrics, indicating minimum and maximum reference values accepted for the metric.

For more details on using metrics to evaluation software product characteristics, please refer to the previous chapter.

Source code metrics apply to different domains (e.g. control flow, data flow, calling relationship) and the range of their scope varies. The scope of a metric designates the element of the source code the metric will apply to.

The different scopes are:

- The *Function scope*: includes functions, procedures, methods, subprograms, tasks, package body, etc. according to the related programming language,

- The *Class scope*: is represented by C++ and Java classes.

- The *Module scope*: is represented by a source code file.

- The *Package scope*: is represented by Java Packages; Packages contain a set of classes.

- The *Application scope*: represented by the set of interrelated software components (i.e. functions, classes) defined in the source code files under analysis.

Not all the standard metrics available in Logiscope *QualityChecker* are presented in the following sub-sections. Only the metrics related to the most well-known theorical approaches (Line Counting, Halstead, Cyclomatic Complexity, MOOD, etc.) are introduced.

In case a metric is only available for some programming languages, this will be stated in the **"Language"** item of the metric specification.

The complete list of standard metrics available in Logiscope *QualityChecker* Ada, C, C++ and Java is provided in the corresponding Reference Manual:

- *IBM Rational Logiscope - QualityChecker & RuleChecker - Ada Reference Manual.*
- *IBM Rational Logiscope - QualityChecker & RuleChecker - C Reference Manual.*
- *IBM Rational Logiscope - QualityChecker & RuleChecker - C++ Reference Manual.*
- *IBM Rational Logiscope - QualityChecker & RuleChecker - Java Reference Manual.*

5.2 Function Scope

5.2.1 Line Counting

For more details on Line Counting Metrics, please refer to §5.6.1.

lc_cline   Total number of lines

Definition   Total number of lines in the function.

lc_cloc   Number of lines of code

Definition   Total number of lines containing executable code in the function.

lc_cblank   Number of empty lines

Definition   Number of lines containing only non printable characters in the function.

lc_ccomm   Number of lines of comments

Definition   Number of lines of comments in the function.
Alias   LCOM

lc_csbra   Number of lines with lone braces

Definition   Number of lines containing only a single brace character : i.e. "{" or "}" in the function.

Languages   C, C++, Java

lc_ccpp   Number of preprocessor statements

Definition   Number of preprocessor directives (e.g. *#include*, *#define*, *#ifdef*) in the function.

Languages   C, C++

lc_bcob Comments blocks before the function

Definition   0 if there is no block of comments used just before a function.
1 if there is a block of comments before; this may indicate that there is comment header before the function.

Example
```
/* this comment is not counted      */
/* as a comment before the function */
int i;
/* this one is counted
   as a comment                     */
/* before the function              */
funct() ;
{
  printf ("----------------------") ;
  printf ("----------------------") ;
}
```
lc_bcob = 1

Languages   C, C++

lc_bcom Number of comments blocks in the function

Definition   Number of comment blocks used between a function header and the closing curly bracket (Blocks of COMments).
Several consecutive comments are counted as a single comment block.

```
funct() ;
{
  /* this is a comment */
  printf ("----------------------") ;
  /* this is a second */
  /* comment          */
  printf ("----------------------") ;
  /* this is a third
     comment          */
}
```

lc_bcom value = 3

Languages   C, C++

5.2.2 Lexical and Syntactic Items

lc_algo   Number of syntactic entities (algorithms)

Definition   Number of syntactic entities inside statements of a function that are not counted as declarations.

Languages   C++, Ada

lc_decl   Number of syntactic entities (declarations)

Definition   Number of syntactic entities in the declaration part of a function.

Languages   C++, Ada

lc_stat   Number of Statements

Definition   Number of executable statements in a function's body

For a detailed specification of executable statements in Ada, C, C++ and Java, refer to the corresponding *IBM RationalIBM Rational Logiscope* manual:

- *QualityChecker & RuleChecker - Ada Reference Manual.*

- *QualityChecker & RuleChecker - C Reference Manual.*

- *QualityChecker & RuleChecker - C++ Reference Manual.*

- *QualityChecker & RuleChecker - Java Reference Manual.*

Alias   STMT

lc_synt Number of syntactic entities

Definition	Number of structures used in a function to represent the program.
Note	**lc_synt** is the sum of l**c_decl** and **lc_algo**.
Languages	C++, Ada

5.2.3 Data Flow

dc_consts Numbers of declared constants

Definition	Number of constants in a function declared by: • the `#define` statement, • variables having a simple type declared as **`const`**, • `enum` elements.
Languages	C++, Ada
Alias	dc_const, NCONST

dc_types Number of declared types

Definition	Number of types declared in a function with the `typedef`, `struct`, `class or enum` statement.
Language	C++, Ada
Alias	dc_type

dc_vars Number of declared variables

Definition	Number of variables declared in a function.
Languages	C++, Ada
Alias	dc_var

dc_excs Number of declared exceptions

Definition	Number of exceptions declared in the exception declarations in a function.
Language	Ada

dc_lvars Number of local variables

Definition	Total number of variables declared in a function (Local VARiables).
Languages	C, C++
Alias	LVAR

dc_clas_var Number of class-type local variables

Definition Number of class type variables which are local to a function. This metric shows a specific type of coupling between classes.

Language C++

Alias LVARop

dc_other_clas_var Number of other class-type local variables

Definition Number of class type variables which are local to a function, where the class is different from the current class. If the function being analyzed is a non-member function, then the value is 0.
This metric is used to compute the **cl_dep_meth** metric.

Language C++

ic_except Number of raised exceptions

Definition Number of exceptions declared by the keyword `throws` in a method.

Languages Java

ic_param Number of parameters

Definition Number of formal parameters.

Alias PARA

ic_parvar Variable number of parameters

Definition Equal to 1 if the function has a variable number of parameters, 0 otherwise.

Language C++

ic_paradd Number of parameters passed by reference

Definition Number of parameters passed by reference of a function. If the function returns a value, then the returned value is considered as a passed by reference parameter.

Language C++

Alias PARAadd

ic_parcl Number of class-type parameters

Definition	Number of class-type parameters of a function. If the function returns a class-type value then the returned value is considered as a class-type parameter. This metric shows a specific type of coupling between classes.
Language	C++
Alias	PARAc

ic_par_othercl Number of other class-type parameters

Definition	Number of a function class-type parameters, where the class is different from the current class.
	If the function being analyzed is a non-member function, then the value is 0.
Language	C++

ic_parval Number of parameters passed by value

Definition	Number of parameters passed by value of a function.
Language	C++
Alias	PARAval

ic_usedp Number of parameters used

Definition	Number of function parameters used in a function body.
	A parameter is said to be used whenever it appears in the function code. Combined with the number of function parameters, this metric is a good indicator of the consistency of the function's interface.
Language	C++
Alias	U_PARA

ic_vare Number of uses of external attributes

Definition	Number of uses of attributes uses defined outside the class.
	An attribute is said to be "external" if it belongs to another class.
	All attribute occurrences are counted.
Language	C++
Alias	VARe

ic_vari   Number of uses of internal attributes

Definition	Number of uses of attributes defined in the class.
	An attribute is said to be "internal" if it belongs to the class of the function being analyzed.
	All attribute occurrences are counted.
Language	C++
Alias	VARi

ic_varpe   Number of distinct uses of external attributes

Definition	Number of distinct times attributes defined outside the class are used.
	An attribute is said to be "external" if it belongs to another class.
	Different uses of the same attribute count for one.
Language	C++
Alias	VAR_PATHSe

ic_varpi   Number of distinct uses of internal attributes

Definition	Number of times the distinct class attributes are used.
	An attribute is said to be "internal" if it belongs to the class of the function being analyzed.
	Different uses of the same attribute count for one.
Language	C++
Alias	VAR_PATHSi

5.2.4 Halstead Metrics

In *"Elements of Software Science"* [HALS, 77], M.H. Halstead developed a theory that deduces a program production and quality characteristics from a small number of parameters: the numbers of operands and operators used in the software component.

The Halstead theory is based on the following four textual metrics:

n1   Number of distinct operators

Definition   Number of different operators used in a function.
Alias   ha_dopt

N1   Total number of operators

Definition   Total number of operators used in a function.
Alias   ha_topt

n2   Number of distinct operands

Definition   Number of different operands used in a function.
Alias   ha_dopd

N2   Total number of operands

Definition   Total number of operands used in a function.
Alias   ha_topd

For a detailed specification of operands and operators in Ada, C, C++ and Java, refer to the corresponding manual:

- *IBM Rational Logiscope - QualityChecker & RuleChecker - Ada Reference Manual,*
- *IBM Rational Logiscope - QualityChecker & RuleChecker - C Reference Manual.*
- *IBM Rational Logiscope - QualityChecker & RuleChecker - C++ Reference Manual.*
- *IBM Rational Logiscope - QualityChecker & RuleChecker - Java Reference Manual.*

Halstead established and later validated many examples based on these metrics, rules that govern a program length, its volume, the implementation level of an algorithm or the language level of the program used.

Then Halstead suggests to use these rules to assess or plan for development time, the time requested to understand software or the number of possible errors.

Halstead metrics available are presented below.

n   Halstead Vocabulary

Definition $n = n1 + n2$

Alias   ha_voc

N   Halstead Program Length

Definition $N = N1 + N2$

Halstead considers *Observed Length* N as the program length observed a posteriori. It takes into account all textual elements present in the code.

Alias   ha_olg

CN   Halstead Estimated length

Definition

$$\hat{N} = n1 * \log_2(n1) + n2 * \log_2(n2)$$

According to Halstead, N and $\hat{N}$ differ by 10% and the correlation coefficient is close to 1. The relation between N (observed length) and $\hat{N}$ (estimated length) seems valid. Then Halstead uses this relation to evaluate other metrics.

Alias   ha_elg

V   Halstead Program Volume

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

According to Halstead, *Program Volume* V corresponds to the minimum number of bits required for program coding.

For each occurrence of N operator or operand which appears in the program, a number v of bits are required to specify it such that $n = 2^v$, thus $v = \log_2 n$. The above formula is deduced from this.

Languages   Ada, C, C++

Alias   ha_vol

L   Halstead Program Level

Definition $L = (2 * n2) / (n1 * N2)$

Alias   ha_lev

Potential Volume V* is the most abridged form into which an algorithm can be expressed. According to Halstead, this most abridged form would use the following (in a language ideally adapted to the algorithm to be translated):

- 2 operators (the name of the function and the assignment operator) and,

- n2* operands where n2* is equal to the number of distinct component input or output parameters.

Program Level L defines a ratio between the *Potential Volume* V* of the algorithm and its actual volume V: L = V* / V

Program Level L represents the implementation level of the algorithm characterized by V*. For the same algorithm, the program level decreases as the program volume increases.

Using the *Potential Volume* evaluation formula produces the above definition.

D   Halstead Program Difficulty

Definition   D = 1/L

Alias   ha_dif

I   Halstead Intelligent Content

Definition   I = L * V
According to Halstead, it represents the algorithm complexity regardless of the language used.
If L is assessed (used to calculate I) to be close to the actual value of *Program Level* (based on *Potential Volume* V*), we can deduce that I is a good estimate value of *Potential Volume* V*.

Alias   ha_int

E   Halstead Mental Effort

Definition   E = V / L
The mental effort E required to both develop and understand a program is expressed by Halstead in "basic reasoning units". It increases with *Program Volume* V and decreases with *Program Level* L.
Halstead showed that a program written in PL/1 requires three times less comprehension effort than a program written in assembly language. He proposed mental effort E as a measurement of program text complexity.

Alias   ha_eff

5.2.5 Structured Programming

In structured programming:

- a function shall have a single entry point and a single exit point,
- each iterative of selective structures shall have a single exit point: i.e. no `goto`, `break`, `continue` or `return` statement in the structure.

Structured programming improves source code maintainability.

ct_bran   Number of destructuring statements

Definition   Number of destructuring statements in a function (`break` and `continue` in loops, and `goto` statements).

ct_break   Number of BREAK and CONTINUE branchings

Definition   Number of `break` or `continue` statements used to exit from loop structures in the function.

`break` statements in `switch` structures are not counted.

Languages   C, C++

ct_exit   Number of out statements

Definition   Number of nodes associated with an explicit exit from a function: *return*, *exit*.

Alias   N_OUT

ct_goto   Number of GOTO statements

Definition   Number of *goto* statements.

Alias   GOTO

ESS_CPX   Essentiel Complexity

Definition   Cyclomatic Number of the "reduced" control graph of the function.
The "reduced" control graph is obtained by removing all structured constructs from the control graph of the function.
A structured contruct is a selective or iterative structure that does not contain a exit statements: `goto`, `break`, `continue` or `return`.

Justification   When the Essentiel Complexity is equal to 1, the function complies with the structured programming rules.
Note that the **ct_exit** and **ct_bran** metrics already provide such an information on the structuring of the function with more details.

5.2.6 Control Graph

ct_andthen   Number of "and_then" operators

Definition   Number of occurrences of the logical operator "`&&`" in the function.

Languages   C

ct_break_inloop   Number of BREAK in loop

Definition   Number of `break` statements used to exit from embedding `loop` structures in the function.

Languages   C

ct_break_inswitch   Number of BREAK in switch

Definition   Number of `break` statements used to exit from embedding `switch` statements in the function.

Languages   C

ct_case   Number of case labels

Definition   Total number of `case` and `default` labels in the function.

Example
```
switch(var)
{
  case A:
  case B: ;
  case C:
     /* A first block of statements */
     i = j + 1;
     break;
  case D:
  case E:
     /* A second block of statements */
     i = k + 1;
     break;
  default:
     /* A third block of statements */
     break;
};
```
ct_case = 6

Languages   C

ct_casepath   Number of case blocks statements

Definition   Total number of blocks of statements in `switch` statements in the function.
Sequential case labels are counted for one block of statements.

```
switch(var)
{
  case A:
  case B: ;
  case C:
     /* A first block of statements */
     i = j + 1;
     break;
  case D:
  case E:
     /* A second block of statements */
     i = k + 1;
     break;
  default:
     /* A third block of statements */
     break;
};
```

ct_casepath = 3

Languages C

ct_continue   Number of CONTINUE statements

Definition   Number of `continue` statements in the function.

Languages   C

ct_dowhile   Number of DOWHILE statements

Definition   Number of `do ... while` statements in the function.

Languages   C

ct_for   Number of FOR statements

Definition   Number of `for` statements in the function.

Languages   C

ct_if   Number of IF statements

Definition   Number of `if` statements in the function.

Languages   C

ct_orelse   Number of "or_else" operators

Definition   Number of occurrences of the logical operator "`||`" in the function.

Languages   C

ct_ternary   Number of ternary operators

Definition   Number of occurrences of the ternary operator "? :" in the function.

Languages   C

ct_nest   Number of nestings

Definition   Maximum nesting level of control structures in a function.

ct_decis   Number of decisions

Definition   Number of selective statements in a function:
`if`, `case`, `switch`, `select`,...

Alias   N_STRUCT

ct_loop   Number of loops

Definition   Number of iterative statements in a function (pre- and post- tested loops):
`for`, `while`, `do while`,

ct_switch   Number of SWITCH statements

Definition   Number of `switch` statements in the function.

Languages   C

ct_while   Number of WHILE statements

Definition   Number of `while` statements in the function.

Languages   C

ct_raise Number of Exception Raises

Definition   Number of occurrences of the *throw* clause within a function body.

Language   C++, Java

Alias   N_RAISE

ct_node Number of nodes

Definition   Number of nodes of a function control graph.

Languages   C++, ADA, Java

Alias   N_NODES

ct_degree Maximum degree

Definition   Maximum number of edges departing from a node.

ct_edge Number of edges

Definition   Number of edges of a function control graph.

Languages   C++, Ada, Java

Alias   N_EDGES

ct_try Number of exceptions handlers

Definition   Number of try blocks in a function.

Languages   C++, Java

Alias   N_EXCEPT

ct_vg Cyclomatic number (VG)

Definition   Cyclomatic complexity number of the control graph of the function. This number depends on the number of nodes of decision in the control graph with the formula:

$$v = 1 + \sum_{NodesOfDecision} n_i - 1$$

where

n_i is the number of edges departing from the node i.

When the control graph has exactly one terminal node (without departing edge) and one origin node (without entering edge), the cyclomatic number is equal to:

$$V(G) = e - n + 2$$

Justification   Whatever the types of structured control used (selections, iterations, branches or sequences) and whatever the way these structures have been assembled (sequentially, nested, structured or not, etc.) the cyclomatic number is the metric used to quantify the complexity of the resulting control structure.

It is therefore a good indicator of the effort the reader must make to understand the function's algorithm and for evaluating the effort that will be required to test its control structure. This metric can also be interpreted to indicate the minimum number of tests cases that will have to be generated to test the function.

Action   A high cyclomatic number is often due to the fact that the function contains too many executable decisions. So the number of decisions will have to be reduced either by subdividing the function or by factorizing any code repetitions that it contains.

This subdivision will result in subroutines being created which will contain the part of the control structure concerning them, thus reducing by as much the original function's control structure. Instead of having a function with a high cyclomatic number, the complexity will be distributed over several functions that have a reasonable cyclomatic number.

Alias VG, ct_cyclo

ct_path Number of paths

Definition Number of non-cyclic execution paths of the control graph of the function.

It is calculated according to the transfers of control induced by the various types of statement. The PATH for a sequence of statements with the same nesting level is the product of each statement's PATH. For nested structures, the sum of the PATH is calculated.

Therefore:
PATH (*if then else endif*) = PATH (body of *then*) + PATH (body of *else*)
PATH (*while do -endwhile*) = PATH (body of *while*) + 1

PATH (*case of - endcase*) = SUM (i=1,n) PATH (body of ist *case*),
 where n is the number of *cases*.
PATH (*sequence*) = 1

Justification PATH gives an idea more accurate than the cyclomatic number of the number of test cases required to fully test a function.
A high number-of-paths value is often due to the fact that the function has too many executable statements. So the number of statements must be reduced either by subdividing the function, or by factorizing any code repetitions it contains.

Alias PATH

DES_CPX Design Complexity

Definition Cyclomatic Number of the "design" control graph of the function.
The "design" control graph is obtained by removing all constructs that do not contain calls from the control graph of the function.

5.2.7 Calling/Called Relations

CALL Number of Calls

Definition	Number of calls in a function.
	Each call to the same function counts for one.
Language	C

dc_calls Number of Direct Calls

Definition	Number of direct calls in a function.
	Different calls to the same function count for one call.
Languages	C, C++
Alias	DRCT_CALLS

dc_calle Number of External Calls

Definition	Number of Calls to Functions Defined outside the Class.
	A function is said to be "defined outside" the class if the function does not belong to the same class as the function being analyzed. If the function being analyzed is a non-member function, then all functions called by the function being analyzed are considered as "defined outside" the class.
	All call occurrences are counted.
Language	C++
Alias	CALLe

dc_calli Number of Internal Calls

Definition	Number of Calls to Functions Defined in the Class.
	A function is said to be "defined in" the class if the function belongs to the same class as the function being analyzed. If the function being analyzed is a non-member function, then there is no function "defined in" the class (the value is 0).
	All call occurrences are counted.
Language	C++
Alias	CALLi

dc_callpe Number of External Direct Calls

Definition	Number of distinct calls to functions defined outside the class of the function being analyzed (see **dc_calle** above).

Different calls to the same function count for one call.

Language C++

Alias CALL_PATHSe

dc_callpi Number of Internal Direct Calls

Definition Number of distinct calls to functions defined in the class of the function being analyzed (see **dc_calli** above).

Different calls to the same function count for one call.

Language C++

Alias CALL_PATHSi

Example:
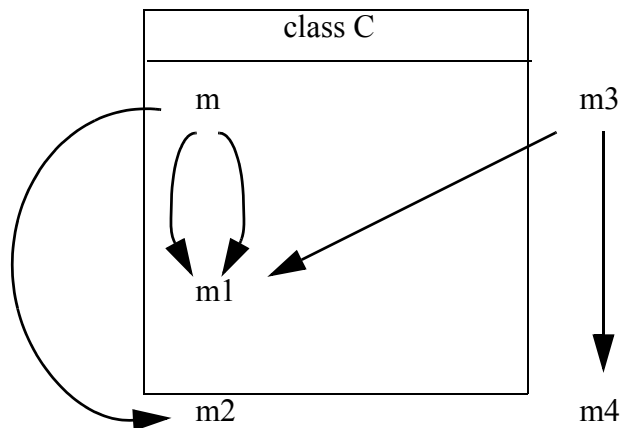
	dc_calle	dc_calli	dc_callpe	dc_callpi
m	$1_{(m2)}$	$2_{(m1)}$	$1_{(m2)}$	$1_{(m1)}$
m3	$2_{(m1, m4)}$	0	$2_{(m1, m4)}$	0

dc_stat_call Number of calls to static members

Definition Number of calls to static member functions in a function.

Language C++

dc_calling Number of callers

Definition Number of functions calling the designated function.

Languages Ada, C

IND_CALLS Number of relative call graph call-paths

Definition   Number of call paths in the relative call graph of the function.

Language   Ada, C, C++,

5.2.8 Relative Call Graph

Mohanty [MOH,76] and SCHUTT [SHT,77] have proposed several complexity metrics derived from the call graph.
At function scope, Logiscope provides the following metrics :

cg_levels   Number of relative call graph levels

Definition   Number of levels of the relative call graph of the function.
Alias   LEVELS

cg_entropy Relative call graph entropy

Definition   This metric proposed by SCHUTT [SHT, 77] applies to the system call graph. It is an indicator of call graph analysability, characterizing both width and depth of the call graph:

$$H(G_A) = \frac{1}{|x|} \sum_{i=1}^{Np} |x_i| \log_2 \frac{|x|}{|x_i|}$$

where |xi| is the number of components in the ith path.
Alias   ENTROPY

cg_ hiercpx Relative call graph hierarchical complexity

Definition   Average number of components per level: i.e. number of components divided by number of levels.
Alias   HIER_CPX

cg_strucpx Relative call graph structural complexity

Definition   Average number of calls per component: i.e. number of calls between components divided by the number of components.
Alias   STRU_CPX

cg_testab   Relative call graph system testability

Definition

$$ST = \frac{1}{Np} \left(\sum_{i=1}^{Np} \frac{1}{TP_i} \right)$$

Np is the number of paths through the system.

TP_i is the testability of the ith call path.

The definition involves the number of paths and the test difficulty level for each path. The result obtained can help to evaluate the software reliability.

Alias   TESTBTY

5.3 Class Scope

5.3.1 Lexical and Syntactic Items

cl_line Number of Lines

Definition	Total number of lines in a class or an interface.
Language	Java

cl_dclstat Number of Declarative Statements

Definition	Number of declarations of fields and methods in a class or an interface.
Language	Java

cl_stat Number of Statements

Definition	Number of statements in all methods and initialization code of a class. This counting of statements and optional parameters *"no_null_stat"* and *"no_decl_stat"* are explained in **lc_stat** in the Function Scope part.
Note	Because the value of the metric **cl_stat** for the class scope depends on the value of **lc_stat** for the method scope, it is strongly recommended to use the same parameters for the two scopes.
Language	Java

cl_bcob Number of comments blocks before the class

Definition	Number of blocks of comments located between a class header and the curly bracket of the previous class or between a class header and the beginning of the file.
Languages	C++
Alias	BCOBc

cl_bcom Number of comments blocks in the class

Definition	Number of blocks of comments in a class. Consecutive comments are counted as belonging to the same block. Comments located outside the class are not counted.
Languages	C++
Alias	BCOMc

cl_comm Number of Comment Lines

Definition   Number of comment lines in a class. Comments located outside the class are not counted.

Languages   Java

5.3.2 Data Flow

cl_base_priv Number of private base classes

Definition   Number of declared classes from which a class inherits, whose names appear after the `private` keyword.

Language   C++

cl_base_prot Number of protected base classes

Definition   Number of declared classes from which a class inherits, whose names appear after the `protected` keyword.

Language   C++

cl_base_publ Number of public base classes

Definition   Number of declared classes from which a class inherits, whose names appear after the `public` keyword.

Language   C++

cl_base_virt Number of virtual base classes

Definition   Number of declared classes from which a class inherits, whose names appear after the `virtual` keyword.

Language   C++

cl_clas_frnd Number of friend classes

Definition   Number of classes declared in a class definition, whose names appear after the `friend` keyword.

Language   C++

cl_interf Number of implemented interfaces

Definition   Number of declared interfaces implemented by a class or extended by an interface.

Language   Java

cl_extend Number of extended classes

Definition Equals 1 if the class extends another class, 0 otherwise.

Language Java

cl_subclass Number of included classes

Definition Number of classes or interfaces declared inside a class or an interface.

Note Anonymous classes are not taken into account.

Language Java

cl_cobc Coupling between classes

Definition Coupling between classes is the sum of:

- the number of inherited classes (see in **in_data_class** Number of Direct Base Classes),
- the number of class type attributes for the class (see **cl_data_class** below),
- two times the number of calls to static member functions for class methods (see in **dc_stat_call** Number of Calls to Static Member Functions),
- two times the number of class-type parameters for the class methods,
- three times the number of class-type local variables for the class methods (see in **dc_clas_var** Number of Class Type Local Variables).

$$\text{cl\_cobc} = \text{in\_dbases} + \text{cl\_data\_class} + \sum_{\text{methods}} (2 \text{\textyen} \text{dc\_stat\_call} + 2 \text{\textyen} \text{ic\_parcl} + 3 \text{\textyen} \text{dc\_clas\_var})$$

Justification **cl_cobc** is an indicator of the degree of dependency of a class. The higher the coupling metric is , the more complex it is to modify the class.

Language C++

Alias COBC, cl_dep_deg

cl_data   Total number of attributes

Definition Total number of data members declared inside a class declaration.

Languages Java

Alias cl_field

cl_data_class Sum of class-type attributes

Definition Number of class-type attributes for the class.

Justification	This metric is used to examine class coupling relationships.
Language	C++
Alias	LACT

cl_data_priv Number of private attributes

Definition	Number of data members declared in the `private` section of a class.
Languages	C++, Java
Alias	LAPI, cl_field_priv

cl_data_prot Number of protected attributes

Definition	Number of data members declared in the `protected` section of a class.
Languages	C++, Java
Alias	LAPO, cl_field_prot

cl_data_publ Number of public attributes

Definition	Number of data members declared in the `public` section of a class.
Languages	C++, Java
Alias	LAPU, cl_field_publ

cl_data_stat Number of static data members

| **Definition** | Number of data members declared after the `static` keyword in a class. |
| **Language** | C++ |

cl_data_inh Number of inherited attributes

| **Definition** | Number of public or protected attributes in the base classes of a class, which are not overridden in that class. |
| **Language** | C++ |

cl_data_final Number of final attributes

Definition	Number of data members declared in a class declaration with the attribute `final`.
Note	For interfaces, **cl_data_final** is equal to **cl_data**.
Language	Java
Alias	cl_field_final

cl_data_const Number of constants

Definition   Number of data members declared in a class declaration with the attributes `final` and `static`.

Note   For interfaces, **cl_data_const** is equal to **cl_data**.

Language   Java

Alias   cl_field_const

cl_data_static Number of class attributes

Definition   Number of data members declared in a class declaration with the attribute `static` and without the `final` attribute.

Note   For interfaces, **cl_data_static** is equal to 0.

Language   Java

Alias   cl_field_static

cl_data_pack Number of attributes in package scope

Definition   Number of data members declared in the class declaration without any of the attributes `private`, `protected` or `public`.

Note   For public classes or interfaces, **cl_data_pack** is equal to 0.

Language   Java

Alias   cl_field_pack

cl_data_nostat Number of instance attributes

Definition   Number of fields declared in a class declaration without attribute `static`.

Note   For interfaces, **cl_data_nostat** is equal to 0.

Language   Java

Alias   cl_field_nostat

cl_dep_meth Number of dependent methods

Definition   Number of methods within the class depending on other classes. A method is said to be dependent if:

- it calls a non-member function or other class methods (see in **dc_calle** Number of Calls to functions Defined outside the Class),
- it uses an attribute which belongs to a different class (see in **ic_vare** Number of Times External Attributes are used),
- it has a class instance parameter which belongs to a different class (see in **ic_par_othercl** Number of Other Class Type Parameters),
- it declares a class instance variable which belongs to a different class (see in **dc_other_clas_var** Number of other Class Type Local Variables).

$$cl\_dep\_meth = \sum_{methods} \begin{cases} 1 \Leftrightarrow dc\_calle + ic\_vare + ic\_par\_othercl + dc\_other\_clas\_var > 0 \\ 0 \qquad otherwise \end{cases}$$

Language   C++

Alias   NMD

cl_rfc Response for a class

Definition   Number of methods that can be invoked in response to a message to an object of the class or by some method in the class.
This includes all methods accessible within the class hierarchy.

Language   C++

cl_type Number of local types

Definition   Number of types declared in a class.

Language   C++

cl_const Number of local constants

Definition   Number of constants declared in a class. Constants are data members declared with the keyword `const`, like `const type name ...`, or `type * const name ...` (constant pointer), or `type C::* const name` (constant pointer to member) for instance (but not pointers to constant).

Language   C++

cl_genp Number of of parameters for templates

Definition   Number of parameters declared in a class for classes that are templates. If cl_genp has the value 0 the class is not a template.

Language   C++

Alias   N-GENC

cl_oper_conv Number of conversion operators

Definition   Number of conversion operators declared in a class declaration.

Language   C++

cl_oper_std Number of standard operators

Definition   Number of operators declared in a class, whose names belong to a certain list being a parameter of the metric (by default, this list is empty).

Language   C++

cl_oper_affc Number of assignment operators

Definition   Number of operators declared in a class, whose names belong to a certain list which is a parameter of the metric (by default, this list contains `"="`, `"+="`, `"-="`, `"*="`, `"/="`, `"%="`, `"^="`, `"&="`, `"|="`, `"<<="`, `">>="`, `"+"`, `"-"`, `"*"`, `"/"` and `"[]"`).

Language   C++

cl_oper_spec Number of special operators

Definition   Number of operators declared in a class, whose names belong to a certain list which is a parameter of the metric (by default, this list contains `"->"`, `"()"`, `","`, `"->*"`, `"new"`, `"delete"`, `"new[]"`, and `"delete[]"`).

Language   C++

5.3.3 Function Aggregates

cl_func Total number of methods

Definition   Total number of methods declared inside a class.

Language   Java

Alias   cl_meth

cl_func_priv Number of private methods

Definition   Number of methods declared in the `private` section of a class.

Languages   C++, Java

Alias   LMPL, cl_meth_priv

cl_func_prot Number of protected methods

Definition   Number of methods declared in the `protected` section of a class.

Languages   C++, Java

Alias   LMPO, cl_meth_prot

cl_func_publ Number of public methods

Definition   Number of methods declared in the `public` section of a class.

Languages   C++, Java

Alias   LMPU, cl_meth_publ

cl_func_virt Number of virtual methods

Definition   Number of methods declared after the `virtual` keyword in a class.

Language   C++

cl_func_pure Number of abstract methods

Definition   Number of methods declared after the `virtual` keyword and followed by
`=0` in a class.
Language   C++
Alias   LMABS

cl_func_cons Number of constant methods

Definition   Number of methods declared after the `const` keyword in a class.
Language   C++

cl_func_inln Number of inline methods

Definition   Number of methods declared after the `inline` keyword in a class.
Language   C++

cl_func_excp Number of methods handling or raising exceptions

Definition   Number of methods declared in a class declaration in which:
- the body of the function is a `try` block, or
- the function body contains a `try` block, or
- exceptions are specified using the `throw` keyword.

Language   C++

cl_func_frnd Number of friend functions

Definition   Number of methods declared after the `friend` keyword in a class.
Language   C++

cl_func_inh Number of inherited methods

Definition   Number of public or protected methods in the base classes of a class,
which are not overridden in that class.
Language   C++

cl_func_abstract Number of abstract methods

Definition   Number of methods declared in a class declaration with the attribute `abstract.`

Note   For interfaces, **cl_func_abstract** is equal to **cl_func**.

Language   Java

Alias   cl_meth_abstract

cl_func_native Number of methods implemented in another language

Definition   Number of methods declared in a class declaration with the attribute `native.`

Note   For interfaces, **cl_func_native** should be 0.

Language   Java

Alias   cl_meth_native

cl_func_pack Number of methods in package scope

Definition   Number of methods declared in a class declaration without any of the attributes `private`, `protected` or `public`.

Note   For public interfaces, **cl_func_pack** is equal to 0.

Language   JAVA

Alias   cl_meth_pack

cl_func_over Number of overridden methods

Definition   Number of inherited methods which a class overrides.

Justification   High values for **cl_func_over** tend to indicate design problems. Subclasses should generally add to and extend the functionality of the parent classes rather than overriding them.

Language   C++

Alias   LMRE

cl_func_static Number of class methods

Definition   Number of methods declared in a class declaration with the attribute `static.`

Note   For interfaces, **cl_func_static** is equal to 0.

The sum of **cl_func_static** and **cl_func_nostat** gives the total number of methods **cl_func**.

Language   Java

Alias   cl_meth_static

cl_func_nostat Number of instance methods

Definition   Number of methods declared in a class declaration without the attribute `static.`

Note   For interfaces, **cl_func_nostat** is equal to **cl_func**.

The sum of **cl_func_static** and **cl_func_nostat** gives the total number of methods **cl_func**.

Language   Java

Alias   cl_meth_nostat

5.3.4 Statistical Aggregates of Function Metrics

cl_fpriv_path Sum of PATH for private class methods

Definition   Sum of non-cyclic execution paths for each class's private methods. This metric is an indicator of the static complexity of the private part of the class.

$$cl\_fpriv\_path = \underset{private}{SUM} (cl\_path)$$

Languages   C++, Java

Alias   LMPIPATH

cl_fprot_path Sum of PATH for protected class methodss

Definition   Sum of non-cyclic execution paths for each class's protected methods. This metric is an indicator of the static complexity of the class protected part.

$$cl\_fprot\_path = \underset{protected}{SUM} (ct\_path)$$

Languages   C++, Java

Alias   LMPOPATH

cl_fpubl_path Sum of PATH for public class methods

Definition   Sum of non-cyclic execution paths for each class's public methods. This metric is an indicator of the static complexity of the public part of the class.

$$cl\_pub\_path = \underset{public}{SUM} (ct\_path)$$

Languages   C++,Java

Alias   LMPUPATH

cl_func_calle Sum of dc_callpe of class methods

Definition   Total number of calls from the class methods to functions defined outside a class (non-member functions or member functions of other classes).

$$cl\_func\_calle = \sum_{methods} dc\_callpe$$

Language   C++

Alias   LMCALL_PATHSe

cl_func_calli Sum of dc_callpi of class methods

Definition   Total number of calls from class methods to member functions of the same class.

$$cl\_func\_calli = \sum_{methods} dc\_callpi$$

Language   C++

Alias   LMCALL_PATHSi

cl_usedp Sum of ic_usedp of class methods

Definition   Total number of parameters used in the class methods.

$$cl\_usedp = \sum_{methods} ic\_usedp$$

Language   C++

Alias   LMU_PARA

cl_data_vare Sum of ic_varpe of class methods

Definition   Total number of times attributes which are external to the class (defined in other classes) are used by the class methods.

$$cl\_data\_vare = \sum_{methods} ic\_varpe$$

Language   C++

Alias   LMVAR_PATHSe

cl_data_vari Sum of ic_varpi of class methods

Definition   Total number of times the class's attributes are used by the class methods.

$$cl\_data\_vari = \sum_{methods} ic\_varpi$$

Language   C++

Alias   LMVAR_PATHSi

The following metrics have been introduced by Shyam R. Chidamber and Chris F. Kemerer in "*A Metrics Suite for Object Oriented Design*" (IEEE Transactions on Software Engineering, vol 20, pp. 476-493, June 1994).

cl_wmc Weighted Methods per Class

Definition   This metric represents the Weighted Methods per Class (WMC). It is the sum of static complexities of class methods. Static complexity is represented in this calculation by the **ct_vg** of functions (see in ct_cyclo Cyclomatic Number).

$$cl\_wmc = \sum_{methods} ct\_vg$$

Justification   The greater the WMC valu is, the more complex, the more difficult to understand and to maintain the class is. Moreover, a high WMC class is most probably specific which limits reuse possibilities.

Languages   C++, Java

Alias   LMVG, cl_cyclo

cl_locm Lack of cohesion of methods

Definition   Percentage of methods that do not access a specific attribute of a class averaged over all attributes in that class.

$$cl\_locm = \frac{\sum_{i=1}^{TA} (1 - Ac(A_i))}{TA}$$

where:

$$Ac(A_i) = \frac{\sum\limits_{i=1}^{TM} \text{is\_accessed}(A_i, M_j)}{TM}$$

and:

$$\text{is\_accessed}(A_i, M_j) = \begin{cases} 1 & \text{if } M_j \text{ accesses } A_i \\ 0 & \text{otherwise} \end{cases}$$

Justification	A low percentage indicates high coupling between methods, which leads to high testing effort (since many methods can affect the same attribute) and potentially low reusability.
Language	C++

5.3.5 Inheritance Tree

in_dbases Number of direct base classes

Definition	Number of classes from which a class directly inherits.
Note	A value of **in_dbases** upper than 1 denotes multiple inheritance.
Justification	If the **in_dbases** value is high, the use of multiple inheritance will be high, increasing class complexity.
Language	C++, Java
Alias	MII, in_dinherits

in_bases Number of base classes

Definition	Number of classes from which a class inherits directly or not If multiple inheritance is not used, the value of **in_bases** is equal to the value of **in_depth**.
Language	C++, Java
Alias	in_inherits

in_depth Depth of the inheritance tree

Definition	Maximum length of an inheritance chain starting from a class.
Languages	C++, Java

in_noc Number of children

Definition	Number of classes which inherit directly from a class.
Justification	The children number of a class is an indicator of the class criticalness within a given system. In fact, more children a class has, more the modifications made to the class will induce changes in the global system.

| Languages | C++, Java |
| **Alias** | NOC, in_dderived |

in_derived Number of derived classes

| **Definition** | Total number of classes which inherit from a class directly or indirectly. |
| **Languages** | C++, Java |

in_reinh Number of classes inherited several times

| **Definition** | Number of classes from which a class inherits several times by different inheritance chains. |
| **Language** | C++ |

5.3.6 Use Graph

There are three kinds of use relationships between classes:

- *characteristic use*: that is when a class C1 derives from a class C2 (inheritance), or when C2 is used for typing a C1 member,

- *contextual use*: that is when C2 is used for typing parameters or the return value of a C1 function member,

- *operational use*: that is when C2 is locally used in a C1 function member body, for typing a local variable, for creating an object (by the way of the `new` instruction), for type conversion (cast), or for accessing a C1 static member directly.

The metrics below take the three kinds of use into account:

cu_level Depth of the Chain of Use

| **Definition** | Maximum length of a chain of use starting from a class (not counting use loop). |
| **Languages** | C++, JAVA |

cu_cdused Number of direct used classes

| **Definition** | Number of classes used directly by a class. |
| **Languages** | C++, JAVA |

cu_cused Number of used classes

| **Definition** | Number of classes used by the current class directly or not. |
| **Languages** | C++, JAVA |

cu_cdusers Number of direct users classes

Definition   Number of classes which use directly a class.

Languages   C++, JAVA

cu_cusers Number of users classes

Definition   Total number of classes which use directly or not a class.

Languages   C++, JAVA

5.4 Module Scope

5.4.1 Lines Counting

For more details on Line Counting Metrics, please refer to §5.6.1

md_blank   **Number of empty lines**

Definition   Number of lines containing only non printable characters in the module.

md_comm   **Number of lines of comments**

Definition   Number of lines of comments in the module.
Alias   LCOM

md_cpp   **Number of preprocessor statements**

Definition   Number of statements computed by the preprocessor (e.g. `#include`, `#define`, `#ifdef`) in the module.
Languages   C,C++

md_line   **Total number of lines**

Definition   Total number of lines in the module.

md_loc   **Number of lines of code**

Definition   Total number of lines containing executable code in the module.

md_sbra   **Number of lines with lone braces**

Definition   Number of lines containing only a single brace character : i.e. "{" or "}" in the module.
Languages   C, C++,Java

md_pro_c: Number of lines in Pro*C

Definition   Total number of lines of PRO*C within the module.
Languages   C++

md_dclstat: Number of declarative statements

Definition   Total number of declarations in the method bodies in the module.

Languages   Java

md_stat: Number of statements

Definition   Total number of executable statements in the method bodies in the module.

Languages   Ada, C++, Java

md_parse: Number of lines not parsed

Definition   Total number of lines not parsed within the module.

Languages   C++, Java

5.4.2 Data Flow

md_consts Number of declared constants

Definition   Number of constants declared in the module.

Languages   Ada, C++

md_excs: Number of declared exceptions

Definition   Total number of exceptions declared in the exception declaration in the module.

Languages   C++

md_types Number of declared types

Definition   Number of types declared in the module.

Languages   Ada, C++

md_vars Number of declared variables

Definition   Number of variables declared in the module.

Languages   Ada, C++

5.4.3 Textual Elements

md_algo Number of syntactic entities (algorithms)

Definition   Number of syntactic entities inside statements that are not counted as declaration in a module.

Languages   Ada, C++

md_decl Number of syntactic entities (declarations)

Definition   Number of syntactic entities in the declaration part of the module (function headers and declaration.

Languages   Ada, C++

md_synt Number of syntactic entities

Definition   Total number of syntactic entities in the module.

Languages   Ada, C++

md_n1 Number of distinct operators

Definition   Number of distinct operators referenced in the module.

Languages   Ada, C++

md_n2 Number of distinct operands

Definition   Number of distinct operands referenced in the module.

Languages   Ada, C++

md_N1 Total number of operators

Definition   Total number of operators referenced in the module.

Languages   Ada, C++

md_N2 Total number of operands

Definition   Total number of operands referenced in the module.

Languages   Ada, C++

md_n Halstead Vocabulary

Definition   Halstead Vocabulary of the module.
$n = n1 + n2$

Languages   Ada, C++

md_N Halstead Program Length

Definition Halstead observed length of the module.
$$N = N1 + N2$$

Languages Ada, C++

md_CN Halstead Estimated Length

Definition Halstead estimated length of the module.

$$\hat{N} = n1 * \log_2(n1) + n2 * \log_2(n2)$$

According to Halstead, N and $\hat{N}$ differ by 10% and the correlation coeffi-

cient is close to 1. The relation between N (observed length) and $\hat{N}$ (esti-
mated length) seems valid. Then Halstead uses this relation to evaluate
other metrics.

Languages Ada, C++

md_V Halstead Program Volume

Definition Halstead Program Volume
$$V = N * \log_2(n)$$

According to Halstead, *Program Volume* V corresponds to the minimum
number of bits required for program coding.
For each occurrence of N operator or operand which appears in the pro-

gram, a number v of bits are required to specify it such that $n = 2^v$, thus
$v = \log_2 n$. The above formula is deduced from this.

Languages Ada, C++

md_L Halstead Program Level

Definition Halstead Program Level
$$L = (2 * n2) / (n1 * N2)$$

Languages Ada, C++

md_D Halstead Program Difficulty

Definition   Halstead Program Difficulty
$$D = 1/L$$

Languages   Ada, C++

md_I Halstead Intelligent Content

Definition   Halstead Intelligent Content
$$I = L * V$$
According to Halstead, it represents the algorithm complexity regardless of the language used.

If L is assessed (used to calculate I) to be close to the actual value of *Program Level* (based on *Potential Volume* V*), we can deduce that I is a good estimate value of *Potential Volume* V*.

Languages   Ada, C++

md_E Halstead Mental Effort

Definition   Halstead Intelligent Content
$$E = V / L$$
The mental effort E required to both develop and understand a program is expressed by Halstead in "basic reasoning units". It increases with *Program Volume* V and decreases with *Program Level* L.

Halstead showed that a program written in PL/1 requires three times less comprehension effort than a program written in assembly language. He proposed mental effort E as a measurement of program text complexity.

Languages   Ada, C++

5.4.4 Interface

md_class Number of classes

Definition   Number of classes declared at the first level of a module.

Languages   Java

md_interf Number of interfaces

Definition   Number of interfaces declared at the first level of a module.

Languages   Java

md_expco Number of exported constants

Definition   Numbers of constants exported by the differents compilation units of the module.

Languages   Ada

md_expex Number of exported exceptions

Definition   Numbers of exceptions exported by the differents compilation units of the module.

Languages   Ada

md_expty Number of exported types

Definition   Numbers of types exported by the differents compilation units of the module.

Languages   Ada

md_expfn Number of exported functions

Definition   C++: Number of non-static global functions defined in the module
Ada: Numbers of functions (packages, subprograms, tasks) exported by the differents compilation units of the module.

Languages   Ada, C++

md_impmo Number of imported modules

Definition   Number of modules included inside a module

Languages   C++

md_import_pack Number of imported packages

Definition   Number of packages appearing in the import statement of a module.
The parameter of the import stement is supposed to be a package name if it is a simple name (without a dot in it) or if it is not used as a type in the module.

Languages   Java

md_import_demd Number of importations on demand

Definition   Number of import statements in a module whose parameter is a generic name (ended by .*).

Languages   Java

md_import_type Number of imported types

Definition   Number of types appearing in the import statements of a module.
The parameter of the import stement is supposed to be a type name if it is not a simple name (with at least a dot in it) or if it is used as a type in the module.

Languages Java

md_with Number of WITH clauses

Definition Numbers of WITH clauses in the module.

Languages Ada

5.5 Package Scope

Metrics at Package scope are only available in Logiscope Java.

5.5.1 Packages Aggregates

pk_line   Number of lines

Definition   Total number of lines in the source files containing the package.

pk_com: Number of lines of comments

Definition   Total number of comment lines in a package.
Comments located outside the package are not counted.

pk_file Number of files

Definition   Total number of files within the package.

pk_pkused Number of imported packages

Definition   Number of imported packages of the package.

5.5.2 Textual Elements

pk_n1   Number of distinct operators

Definition   Number of distinct operators referenced in the package.

pk_n2   Number of distinct operands

Definition   Number of distinct operands referenced in the package.

pk_N1   Total number of operators

Definition   Total number of operators referenced in the package.

pk_N2   Total number of operands

Definition   Total number of operands referenced in the package.

pk_cpx_max Maximum size of statements

Definition   Maximum number of operands and operators in a statement of the package.

5.5.3 Statistical Aggregates of Class Metrics

pk_class Number of classes

Definition   Total number of classes declared in the package.
Nested classes are counted.

pk_interf Number of interfaces

Definition   Total number of interfaces declared in the package.

pk_const Number of constants

Definition   Total number of constants declared in the classes of the package.

pk_data Number of data

Definition   Total number of data declared in the classes of the package.

pk_data_priv Number of private data

Definition   Total number of data explicitly declared with the "private" keyword in
the classes of the package.

pk_data_prot Number of protected data

Definition   Total number of data explicitly declared with the "protected" keyword in
the classes of the package.

pk_data_publ Number of public data

Definition   Total number of data explicitly declared with the "public" keyword in
the classes of the package.

pk_data_stat Number of static data

Definition   Total number of data explicitly declared with the "static" keyword in the
classes of the package.

pk_except Number of raised exceptions

Definition   Total number of exceptions declared by the keyword "throw" in the
method declaration of the package.

pk_raise Number of raising an exceptions raises

Definition   Total number of occurrences of "throw" in the classes of the package.

pk_try Number of exception handlers

Definition   Total number of occurrences of "try" blocks in the classes of the package.

pk_type Number of public classes

Definition   Total number of public classes of the package.

5.5.4 Statistical Aggregates of Function Metrics

pk_func Number of functions

Definition   Total number of functions declared in the classes of the package

pk_func_priv Number of private functions

Definition   Total number of functions explicitely declared with the "private" key-word in the classes of the package.

pk_func_prot Number of protected functions

Definition   Total number of functions explicitely declared with the "protected" key-wordin the classes of the package.

pk_func_publ Number of public functions

Definition   Total number of functions explicitely declared with the "public" keyword in the classes of the package.

pk_func_stat Number of static functions

Definition   Total number of functions explicitely declared with the "static" keyword in the classes of the package.

pk_func_abstract Number of abstract functions

Definition   Total number of abstract functions in the classes of the package.

pk_func_used Number of called functions

Definition   Number of calls of functions by a function declared in the classes of the package.

pk_func_used_max Maximum number of called functions

Definition Maximum number of calls of functions by a function declared in the classes of the package.

pk_levl Sum of nested levels

Definition Sum of nested levels (ct_nest) in the functions declared in the classes of the package.

pk_levl_max Maximum nested levels

Definition Maximum number of nested levels (ct_nest) in a function declared in the classes of the package.

pk_path Sum of non-cyclic paths

Definition Sum of Non-Cyclic Paths (ct_path) in the functions declared in the classes of the package.

pk_path_max Maximum number of non-cyclic paths

Definition Maximum number of Non-Cyclic Paths (ct_path) in a function declared in the classes of the package.

pk_param Sum of parameters

Definition Sum of the number of formal parameters (ic_param) in the functions declared in the classes of the package.

pk_param_max Maximum number of parameters

Definition Maximum number of formal parameters (ic_param) in a function declared in the classes of the package.

pk_stmt Sum of executable statements

Definition Sum of executable statements (lc_stat) in the functions declared in the classes of the package.

pk_stmt_max Maximum number of statements

Definition Maximum number of executable statements (lc_stat) in a function declared in the classes of the package.

pk_vg Sum of Cyclomatic Numbers

Definition   Sum of Cyclomatic Numbers (ct_vg) of the functions declared in the classes of the package.

pk_vg_max Maximum Cyclomatic number

Definition   Maximum Cyclomatic Number (ct_vg) in a function declared in the classes of the package.

5.5.5 Inheritance

pk_extend: Number of inheritance using extend

Definition   Number of classes referenced in the "extend" directives of the classes in the packages.
If a class is referenced several times, it is counted several times.

pk_implement: Number of inheritance using implement

Definition   Number of classes referenced in the "implement" directives of the classes in the packages.
If a class is referenced several times, it is counted several times.

pk_inh_levl: Sum of depth of the inheritance tree

Definition   Sum of the depth of the classes in the inheritance tree of the classes declared in the package.

pk_inh_levl_max: Depth of the inheritance tree

Definition   Maximum length of an inheritance chain starting from a class declared in the package.

5.6   Application Scope

The application is determined by the list of source files specified in the Logiscope projet. Consequently, the value of metrics presented in this section may change depending on the source files analyzed of not. It is therefore recommended to use these metrics results exclusively for a complete application or for a coherent subsystem.

5.6.1 Line Counting

Lines Of Code (LOC) are often used for quantifying the productivity of the development and maintenance phases ... even if such an approach is clearly subject to caution.

Logiscope *QualityChecker* provides many basic metrics that can be combined together to provide the appropriate LOC counting regarding the context or the applicable standard, for instance:

- *Physical LOC*: all lines in a source file (see ap_sline) where you may or not remove substract the empty lines (see ap_sblank);

- *Physical executable LOC*: all lines containing executable code (see ap_sloc): i.e. all lines except lines of comments (see ap_scomm) and empty lines (see ap_sblank);

- *Effective LOC*: same as above but removing also simple lines of code: i.e. lines containing only braces for presentation purposes (see ap_ssbra) and preprocessing directives (see ap_scpp).

- *Logical LOC*: not impacted by the the way the code is presented but only taken into account syntactical executable statements as specified in the programming language (see ap_stat).

The main basic metrics available in Logiscope *QualityChecker* for line counting are :

ap_sline   Total number of lines

Definition   Total number of lines in the application source files.

ap_sloc   Number of lines of code

Definition   Total number of lines containing executable in the application source files.

ap_sblank   Number of empty lines

Definition   Total number of lines containing only non printable characters in the application source files.

ap_scomm   Total number of lines of comments

Definition   Number of lines of comments in the application source files.

ap_scpp   Number of preprocessor statements

Definition   Number of preprocessor directives (e.g. *#include*, *#define*, *#ifdef*). in the application source files.

Languages   C, C++

md_ssbra   Number of lines with lone braces

Definition   Number of lines containing only a single brace character : i.e. "{" or "}" application source files.

Languages   C, C++, Java

5.6.2 Function Aggregates

ap_func **Number of application functions**

Definition Number of functions in the application. The application is defined by the list of analyzed files. For C++ language, this metric counts non-member functions only.

Alias LMA

ap_interf_func Number of application interface functions

Definition Number of interface functions in the application.

Languages Java

ap_mdf **Sum of application defined methods**

Definition Number of methods defined in the application.

Language C++

Alias NMM

ap_npm **Sum of application public methods**

Definition Number of public methods in the application.

Language C++

ap_nmm **Sum of application member functions**

Definition Number of member functions in the application. The application is defined by the list of analyzed files.

Language C++

Alias NMM

ap_line **Number of function lines**

Definition Sum of number of lines (i.e. lc_line) of all the functions defined in the application.

$$ap\_line = \sum_{functions} lc\_line$$

ap_stat **Number of statements**

Definition Sum of executables statements: (i.e. lc_stat) of all the functions defined in the application.

$$ap\_stat = \sum_{functions} lc\_stat$$

ap_vg   Sum of cyclomatic numbers of the functions

Definition Sum of Cyclomatic Number: i.e. ct_vg of all the functions defined in the application.

$$ap\_vg = \sum_{functions} ct\_vg$$

Alias VGA, ap_cyclo

5.6.3 Sum of Class Metrics

ap_cbo   Coupling between objects

Definition Sum of relationships from class to class other than inheritance relationships:
- **cl_func_calle** Sum of **dc_callpe** from class methods
- **cl_data_class** Sum of class-type attributes.

$$ap\_cbo = \sum_{classes} (cl\_func\_calle + cl\_data\_class)$$

Language C++
Alias CBO

ap_clas   Number of application classes

Definition Number of classes in the application.
Languages C++, Java
Alias LCA

5.6.4 MOOD

The MOOD (Metrics for Object Oriented Design) set of metrics described in this chapter has been introduced by Fernando Brito e Abreu in "*Object-Oriented Software Engineering: Measuring and Controlling the Development Process*" (Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994).

Their definitions have been refined since their first introduction. The MOOD metrics conform to last definitions and corresponding C++ bindings described in "*Evaluating the Impact of Object-Oriented Design on Software Quality*" (Proceedings of the Third International Software Metrics Symposium, IEEE, Berlin, Germany, March 1996).

This approach is only available for Logiscope C++ projects.

ap_mhf   Method hiding factor

Definition

$$ap\_mhf = \frac{\sum_{i=1}^{TC} \left[\sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi})) \right]}{\sum_{i=1}^{TC} M_d(C_i)}$$

where:

$$V(M_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

and:

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 & \begin{cases} j \neq i \\ C_j \text{ may call } M_{mi} \end{cases} \\ 0 & otherwise \end{cases}$$

The MHF numerator is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of total classes from which this method is not visible.

The MHF denominator is the total number of methods defined in the project.

Alias   MHF

The following C++ bindings are used to compute this metric:

MOOD		C++
TC	total classes	total number of classes
	methods	constructors; destructors; function members; operator definitions
$M_d(C_i)$	**methods defined** (not inherited)	all methods declared in the class including virtual (deferred) ones
$V(M_{mi})$	**visibility - percentage of total classes from which the method M_{mi} is visible**	= 1 for methods in public clauses; = 0 for methods in private clauses; = $DC(C_i)/(TC-1)$ for methods in protected clauses ($DC(C_i)$ = descendants of C_i)

ap_ahf   Attribute hiding factor

Definition

$$ap\_ahf = \frac{\sum_{i=1}^{TC}\left[\sum_{m=1}^{A_d(C_i)}(1 - V(A_{mi}))\right]}{\sum_{i=1}^{TC} A_d(C_i)}$$

where:

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1}$$

and:

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 \ntieq \begin{cases} j \neq i \\ C_j \text{ may reference } A_{mi} \end{cases} \\ 0 \quad \text{otherwise} \end{cases}$$

The AHF numerator is the sum of invisibilities of all attributes defined in all classes. The invisibility of an attribute is the percentage of total classes from which this attribute is not visible.

The AHF denominator is the total number of attributes defined in the project.

Alias   AHF

The following C++ bindings are used to compute this metric:

MOOD		C++
$A_d(C_i)$	**attributes defined** (not inherited)	data members

$V(A_{mi})$	**visibility - percentage of total classes from which the attribute A_{mi} is visible**	= 1 for attributes in public clauses; = 0 for attributes in private clauses; = $DC(C_i)/(TC-1)$ for attributes in protected clauses ($DC(C_i)$ = descendants of C_i)

ap_mif Method inheritance factor

Definition

$$ap\_mif = \frac{\displaystyle\sum_{i=1}^{TC} M_i(C_i)}{\displaystyle\sum_{i=1}^{TC} M_a(C_i)}$$

where:

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

The MIF numerator is the sum of inherited methods in all classes of the project.

The MIF denominator is the total number of available methods (locally defined plus inherited) for all classes.

Alias   MIF

The following C++ bindings are used to compute this metric:

MOOD		**C++**
$M_a(C_i)$	**available methods**	function members that can be invoked in association with C_i
$M_d(C_i)$	**methods defined**	function members declared within C_i
$M_i(C_i)$	**inherited methods**	function members inherited (and not overridden) in C_i

ap_aif Attribute inheritance factor

Definition

$$ap\_aif = \frac{\displaystyle\sum_{i=1}^{TC} A_i(C_i)}{\displaystyle\sum_{i=1}^{TC} A_a(C_i)}$$

where:

$$A_a(C_i) = A_d(C_i) + A_i(C_i)$$

The AIF numerator is the sum of inherited attributes in all classes of the project.

The AIF denominator is the total number of available attributes (locally defined plus inherited) for all classes.

Alias   AIF

The following C++ bindings are used to compute this metric:

MOOD		C++
$A_a(C_i)$	**available attributes**	data members that can be invoked associated with C_i
$A_d(C_i)$	**attributes defined**	data members declared within C_i
$A_i(C_i)$	**inherited attributes**	data members inherited (and not overridden) in C_i

ap_pof   Polymorphism factor

Definition

$$ap\_pof = \frac{\sum\limits_{i=1}^{TC} M_o(C_i)}{\sum\limits_{i=1}^{TC} [M_n(C_i) \yen DC(C_i)]}$$

where:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

The POF numerator is the sum of overriding methods in all classes. This is the *actual number of possible different polymorphic situations*. Indeed, a given message sent to a class can be bound, statically or dynamically, to a named method implementation. The latter can have as many shapes (morphos) as the number of times this same method is overridden (in that class's descendants).

The POF denominator represents the *maximum number of possible distinct polymorphic situations* for that class as the sum for each class of the number of new methods multiplied by the number of descendants. This value would be maximum if all new methods defined in each class would be overridden in all of their derived classes.

Alias   POF

The following C++ bindings are used to compute this metric:

MOOD		C++
$DC(C_i)$	**descendants count**	number of classes descending from C_i
$M_n(C_i)$	**new methods**	function members declared within C_i that do not override inherited ones
$M_o(C_i)$	**overriding methods**	function members declared within C_i that override (redefine) inherited ones

ap_cof   Coupling factor

Definition

$$ap\_cof = \frac{\displaystyle\sum_{i=1} \left\lfloor \sum_{j=1} is\_client(C_i, C_j) \right\rfloor}{TC^2 - TC}$$

where:

$$is\_client(C_i, C_j) = \begin{cases} 1 \Leftrightarrow C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 \quad \text{otherwise} \end{cases}$$

The COF denominator stands for the *maximum possible number of couplings* in a system with TC classes.

The client-supplier relation (represented by $C_c \Rightarrow C_s$) means that C_c (*client* class) contains *at least one* non-inheritance reference to a feature (method or attribute) of class C_s (*supplier* class). The COF numerator then represents the *actual number of couplings not imputable to inheritance*.

Alias   COF

Client-supplier relations can have several shapes:

Client-supplier shapes	C++
regular message passing	call to the interface of a function member in another class
"forced" message passing	call to a visible or hidden function member in another class by means of a friend clause
object allocation and deallocation	call to a class constructor or destructor
semantic associations among classes with a certain arity (e.g. 1:1, 1:n or n:m)	reference to a supplier class as a data member or as a formal parameter in a function member interface

5.6.5 Application Call Graph

ap_cg_cycle Call Graph recursions

Definition   Number of recursive paths in the call graph for the application's functions. A recursive path can be for one or more functions.

Justification   Excessive use of recursiveness increases the global complexity of the application and may diminish system performances.

Alias   GA_CYCLE

ap_cg_edge Number of Edges in the Call graph

Definition   Number of edges in the call graph of application functions.

Alias   GA_EDGE

ap_cg_levl Number of Levels in the Call graph

Definition   Depth of the Call Graph: number of call graph levels.

Justification   Too many call graph levels indicates a strong hierarchy of calls among system functions. This may be due to incorrectly implemented object-coupling relationships.

Alias   GA_LEVL

ap_cg_maxdeg Maximum of Calling/Called

Definition   Maximum number of calling/called for nodes in the call graph of application functions.

Languages   C, ADA

Alias   GA_MAXDEG

ap_cg_maxin Maximum of Calling

Definition   Maximum number of "callings" for nodes in the call graph of Application functions.

Alias   GA_MAX_IN

ap_cg_maxout Maximum of Called

Definition   Maximum number of called functions for nodes in the call graph of Application functions.

Alias   GA_MAX_OUT

ap_cg_node Number of Nodes in the Call graph

Definition   Number of nodes in the call graph of Application functions. This metric cumulates Application's member and non-member functions as well as called but not analyzed functions.

Alias   GA_NODE

ap_cg_root Number of Roots

Definition   Number of roots functions in the call graph of Application functions.

Alias   GA_NSP

ap_cg_leaf Number of Leaves

Definition   Number of functions executing no call. In other words, number of leaves nodes in the call graph of Application functions.

Alias   GA_NSS

Example:

Application call graph

  : x calls y
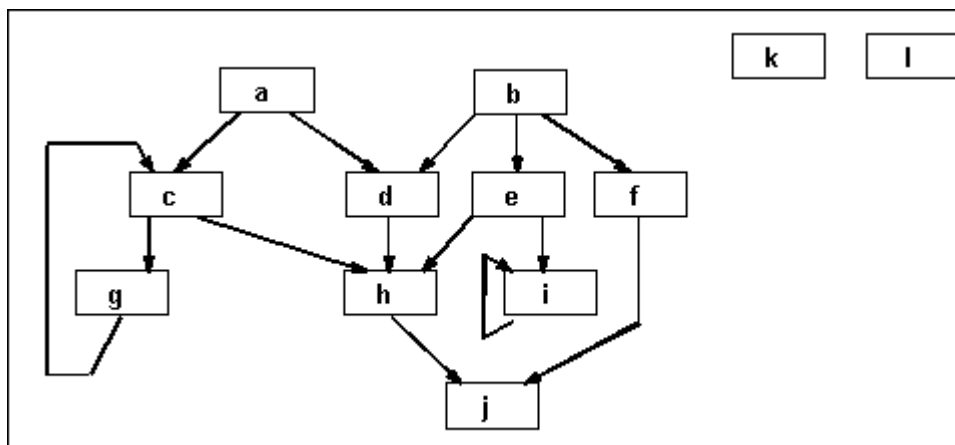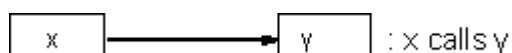
j : called but not analyzed component.

Call graph metric values:

```
ap_cg_cycle = 2 (c,g) (i)
ap_cg_edge = 14
ap_cg_levl = 4
ap_cg_maxdeg = 4 (h or c)
ap_cg_maxin = 3 (h)
ap_cg_maxout = 3 (b)
ap_cg_node = 12 (a, b, c, d, e, f, g, h, i, j, k, l)
ap_cg_root = 4 (a, b, k, l)
ap_cg_leaf = 3 (j, k, l)
```

5.6.6 Inheritance Tree

ap_inhg_cpx Inheritance tree hierarchical complexity

Definition   The complexity of the inheritance tree is defined as a ratio between:
- the sum for all of the graph levels of the number of nodes on the level times the level weight index,
- the number of graph nodes.
- Basic classes are on the top level and leaf classes on the lower levels

$$ap\_inhg\_cpx = \frac{SUM\,(N*i)}{SUM\,(N)}$$

where N is the number of nodes for level i.

Justification   This metric is an indicator of the graph structure:
- a tall and narrow graph has an **ap_inhg_cpx** value that goes to ∞,
- a graph as wide as long has an **ap_inhg_cpx** value that goes to 2,
- a graph as wide and long has an **ap_inhg_cpx** value that goes to 1.

The purpose of this metric is to measure the hierarchical complexity of a graph.
An **ap_inhg_cpx** value greater than 2 indicates that the graph hierarchy is too complex.

Languages   C++, JAVA

Alias   GH_CPX

ap_inhg_edge Number of Edges in the Inheritance Graph

Definition   Number of inheritance relationships in the application.

Languages   C++, JAVA

Alias   GH_EDGE

ap_inhg_levl Number of Levels in the Inheritance Graph

Definition   The Depth of the Inheritance Tree (DIT) is the number of classes in the longest inheritance link.

Justification   The greater the **ap_inhg_levl** value is, the greater is the number of inherited functions and the more complex will be the application.

Languages   C++, JAVA

Alias   GH_LEVL

ap_inhg_maxdeg Maximum Number of Derived/Inherited Classes

Definition   Maximum number of inheritance relationships for a given class. This metric applies to the Application's inheritance graph.

Languages   C++, JAVA

Alias   GH_MAX_ DEG

ap_inhg_maxin Maximum Number of Derived Classes.

Definition   Maximum number of derived classes for a given class in the inheritance graph.

Languages   C++, JAVA

Alias   GH_MAX_ IN

ap_inhg_maxout Maximum Number of Inherited Classes.

Definition   Maximum number of inherited classes for a given class in the inheritance graph.

Languages   C++, JAVA

Alias   GH_MAX_ OUT

ap_inhg_node Number of Classes in the Inheritance Graph

Definition   Number of classes present in the inheritance graph.

Languages   C++, JAVA

Alias   GH_NODE

ap_inhg_leaf Number of Leaves Classes

Definition   Number of leaves classes in the application. A class is said to be a leaf class if it has no child class.

Languages   C++, JAVA

Alias   GH_NSP

ap_inhg_root Number of Basic Classes

Definition   Number of basic classes in the application. A class is said to be basic if it does not inherit from any other class.

Languages   C++, JAVA

Alias   GH_NSS

ap_inhg_pc Protocol Complexity of the Inheritance Graph

Definition Inheritance graph depth times the maximum number of functions in a class of the inheritance graph over the total number of functions in the inheritance graph classes

$$ap\_inhg\_pc = ap\_inhg\_levl \times \frac{MAX\ (LMPI + LMPO + LMPU)}{SUM\ (LMPI + LMPO + LMPU)}$$

Languages C++, JAVA
Alias GH_PC

ap_inhg_uri Number of Repeated Inheritances

Definition Repeated inheritances consist in inheriting twice from the same class. The number of repeated inheritances is the number of inherited class couples leading to a repeated inheritance.

Justification Repeated inheritance is a cause of complexity and naming conflict in cases of functions inherited several times. Nevertheless, in certain cases, repeated inheritance can be useful but should not be used excessively.

Languages C++, JAVA
Alias GH_URI

Example
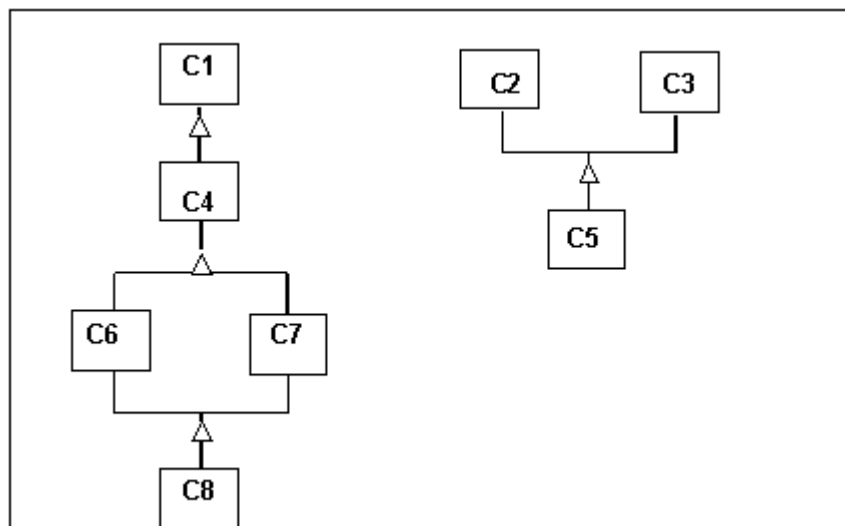
Inheritance graph of the application being analyzed

x◁—y: y inherits from x

ap_inhg_uri	= 1 (C8 inherits from C4 via C6 and C7)
ap_inhg_maxdeg	= 3 (C4)
ap_inhg_maxout	= 2 (C8 or C5)
ap_inhg_maxin	= 2 (C4)
ap_inhg_root	= 3 (C1, C2, C3)
ap_inhg_leaf	= 2 (C8, C5)
ap_inhg_edge	= 7
ap_inhg_node	= 8
ap_inhg_levl	= 4
ap_inhg_cpx	$= \dfrac{(2 \times 1) + (4 \times 2) + (1 \times 3) + (1 \times 4)}{2 + 4 + 1 + 1}$

Chapter 6

Project Configuration Files

6.1   Quality Model

This section describes the internal structure of the Quality Model or Reference file used by Logiscope *QualityChecker* to assess quality using source code metrics (cf. Chapter 4).

6.1.1 Overall Structure

The # character at the beginning of a line indicates that the line is a comment.

The structure of a Quality Model file is as follows:

/language
    # identification of the related programming language:
    # *language*=Ada | C | C++ | Java
MD
    # definitions of user metrics
ME
    # selection of metrics that are actually calculated, definition of their limit values and out-of-bounds messages
MC
    # definitions of quality criteria and categories
BQ
    # definitions of quality factors and categories

6.1.2 *MD* Section: User Defined Metric Specification

Notes

New metrics are defined using expressions combining:
- basic metrics (see Measuring complexity),
- previously user defined metrics.
The maximum number of metrics is 500.

Metric syntax

"*description*" **:** *mnemonic* [= *expression*] [{ *presentation* }]

description

    brief metric syntax description.

mnemonic

    name of the metric (letters, numbers and underscore "_") used whenever the metric is referred to (definition of other metrics, criteria, etc.).

expression

    arithmetical expression using operators and operands.

The following operators are possible:

    + binary addition

    – binary subtraction

    * multiplication

    / division

    ** exponentiation (x**y means x raised to the power of y)

    MOD modulo (x MOD y is equal to the remainder of the division of x by y)

    LOG Neperian logarithm

    LOG2 base 2 logarithm

    MIN(x,y) includes the minimal value between x and y

    MAX(x,y) includes the maximal value between x and y

Possible operands are numeric constants or metric names.

    presentation

    presentation of the metric (free syntax) that is displayed on the righthand side of the double Properties box.

Example

For example, the "comments frequency" metric can be defined by combining the "number of blocks of comments" and "number of statements" basic metrics as follows :
COMF = (lc_bcom + lc_bcob) / (lc_stat)

```
*MD*
```

```
"COMments Frequency" :COMF = (lc_bcom + lc_bcob) / (lc_stat)
```

6.1.3 *ME* Section: Metric Threshold Specification

The ***ME*** section of the Quality Model file is used to set metric limits/thresholds.

Metric thresholds represent the value interval in which the metric value is deemed to be acceptable.

*Note: User defined Metrics referenced in this paragraph must be first defined in the ***MD*** paragraph.*

Metrics definition syntax

mnemonic format {min|-1} {max|+1}
>   mnemonic
>>   name of the metric.
>   format
>>   one of the following formats:
>>   I: integer,
>>   F: floating real (e.g.: 12.34).
>   min
>>   lower limit (integer or real).
>>   If no lower limit is required, enter $-\infty$ (which stands for -1).
>   max
>>   upper limit (integer or real).
>>   If no upper limit is required, enter $+\infty$ (which stands for +1).

*NoteIn order to display the metric's value in the related text, use the **$value** symbol.*

Example

```
*ME*
    COMF            F     0.20    1.00
    VG              I     1       15
    PATH            I     1       80
```

6.1.4 *MC* Section: Quality Criteria Specification

The **MC** section of the Quality Model file is used to define quality criteria and specify their:

- name and comments,
- calculation formula,
- categories or rating levels.

Reminder

A quality criterion has a name which identifies a specific formula based on a combination of weighted metrics.

Each criterion-specific formula is calculated either according to limit values for metrics used in the formula or on the basis of the metric value (whenever the VAL operator is used).

Categories have to be defined for each criterion. Each component enters into one of the criterion's categories, as a function of the combination of its weighted metrics.

NotesThe maximum number of criteria is 100.

*TMetrics referenced in this paragraph must be first defined in the **MD** and **ME** paragraph.*

Criterion definition syntax

criterion = formula$_?$[{ presentation }]
categoryname min max f_weight
categoryname min max f_weight
...

> criterion
>
> Name of the quality criterion (letters, numbers and underscore "_").

> formula
>
> A criterion formula is an arithmetical expression using operators and operands:
> The following are possible operators:
> +   binary addition
> -   binary subtraction
> *   multiplication
> /   division
> **   exponentiation (x**y means x raised to the power of y)
> MOD   modulo (x MOD y is equal to the remainder of the division of x by y)
> LOG   Neperian logarithm
> LOG2   base 2 logarithm
> VAL   operand value
> The possible operands are numeric constants or metric names.

{ *presentation}*

    Presentation of the criterion (free syntax) which is displayed in the righthand side of the double Properties box.

categoryname

    Name of the criterion's category (letters, numbers and underscore "_")

    A category represents the diagnosis associated to a factor value. A category is a range of values for which the diagnosis is identical.

min

    Lower limit (inclusive) for the category concerned

max

    Upper limit (exclusive except for the last category) for the category concerned

f_weight

    Quality factor coefficient associated with the category (integer or real) used to establish quality factors.

    The third value in the definition of a criteria category corresponds to the Quality factor coefficient. This value is used to calculate the overall quality assessment of factors (see *Quality model - File - Defining factors*).

Note: *One category's lower value must be the next category's upper value. Minimal and maximal values can be integers or reals where −∞ and +∞ represent respectively "minus the infinite" and "plus the infinite".*

Evaluating the quality criterion

Components are classified according to the following process:

- If there is no VAL operator, for any given component, if the value of a metric is within its acceptable interval, the metric is associated with the value 1, otherwise it is associated with 0. If there is a VAL operator, for any given component, the value taken into account is the metric's value.

- this type of evaluation is performed on all the metrics entering into the definition of the criterion.

- the expression defining the criterion is calculated and gives the component the value for the criterion concerned.

- this value determines the component's category.

Example

MC

TESTABILITY =	40*VG +	40*LEVL	+20*RETU
ACCEPTED	99	100	0
RESTRUCTURE	80	99	30
SUBDIVIDE	20	80	10
REWRITE	0	20	0

Interpretation of the example

The TESTABILITY criterion is based on whether or not the following 3 metrics are within their acceptable interval:

- VG (cyclomatic number), with a weight coefficient of 40,
- LEVL (number of levels) with a weight coefficient of 40,
- RETU (number of exit points) with a weight coefficient of 20.

The table below gives all possible combinations for the 3 metrics (2^3), and the number of points obtained for each combination. The maximum number of points is 100.

VG	LEVL	RETU	total points	Criterion category
no	no	no	0	REWRITE
no	no	yes	20	SUBDIVIDE
no	yes	no	40	SUBDIVIDE
no	yes	yes	60	SUBDIVIDE
yes	no	no	40	SUBDIVIDE
yes	no	yes	60	SUBDIVIDE
yes	yes	no	80	RESTRUCTURE
yes	yes	yes	100	ACCEPTED

Total points obtained indicates the category that the component belongs to for the given criterion. In our example, 60 enters into the SUBDIVIDE category ([20, 80[) for the TESTABILITY criterion.

Example

MC
    TESTABILITY = 4*VAL(VG + 4*VAL(LEVL) + 2*VAL(RETU)
    VERY POOR 226 +oo
    POOR 168 226
    AVERAGE 126 168
    CORRECT 56 168
    EXCELLENT 0 56

Interpretation of the example

The TESTABILITY criterion is calculated from the value of the `VG`, `LEVL` and `RETU` metrics. The following table gives an example of definitions of categories associated to the criterion.

VG	LEVL	RETU	total points	Criterion category
no	no	no	0	REWRITE
no	no	yes	20	SUBDIVIDE
no	yes	no	40	SUBDIVIDE
no	yes	yes	60	SUBDIVIDE
yes	no	no	40	SUBDIVIDE
yes	no	yes	60	SUBDIVIDE
yes	yes	no	80	RESTRUCTURE

Consider that an EXCELLENT component is a component whose VG value is included between 0 and 3 and whose number of inputs is included between 0 and 2, the component value for the TESTABILITY criterion will be included between 0 and 56.

6.1.5 *BQ* Section: Quality Factors Specification

The ***BQ***section of the Quality Model file is used to define factors and specify their:

- name and comments,
- calculation formula,
- categories or rating levels.

Reminder

Within a given software quality system hierarchy, the top level attribute for quality assessment is the quality factor.

A quality factor formula is evaluated by means of the quality criteria composing it.

A quality factor has a name which identifies a specific formula based on a combination of criteria.

Each factor-specific formula is calculated either according to the value associated with the criteria used in the formula or on the basis of the criterion value (whenever the VAL operator is used).

The user defines the quality factor as a set of categories that represent the extent to which the factor has been fulfilled. Components are classified according to the criteria analysis.

A weight coefficient is associated with each criterion category when it is defined which indicates the contribution it makes towards the factor. All the defined criteria thus contribute towards the evaluation of the factor (except if the weight coefficient is zero).

Categories have to be defined for each factor.

NotesThe maximum number of factors is 50.

*The criteria referenced in this paragraph must be first defined in the ***MC*** paragraph.*

Factor syntax

factor[:scope]?=?formula₂[{ presentation }]
categoryname?min?max
categoryname?min?max

...

factor

name of the quality factor (letters, numbers and underscore "_").

scope: component | class | application

NoteHere, `component` stands for a member or a non-member function.

formula

A factor formula is an arithmetical expression using operators and operands.
The following are possible operators:

+	binary addition
-	binary subtraction
*	multiplication
/	division
**	exponentiation (x**y means x raised to the power of y)
MOD	modulo (x MOD y is equal to the remainder of the division of x by y)
LOG	Neperian logarithm
LOG2	base 2 logarithm
VAL	operand value

The possible operands are numeric constants or criteria names.

{ *presentation}*

presentation of the factor (free syntax).

categoryname

Name of the factor's category (letters, numbers and underscore "_")
A category represents the diagnosis associated to a factor value. A category is a range of values for which the diagnosis is identical.

min

Lower limit (inclusive) for the category concerned.

max

Upper limit (exclusive except if 100) for the category concerned.

NoteOne category's lower value must be the next category's upper value. Minimal and maximal values can be integers where −oo and +oo represent respectively "minus the infinite" and "plus the infinite".

Evaluating the quality factor

Components are classified according to the following process:

- If there is no VAL operator, for any given component, the criterion value taken into account is *f_weight* associated with the component category for the criterion. If a VAL operator is specified, the value taken into account is the result of the evaluation of the criterion's formula.

- this type of evaluation is performed on all the criteria entering into the definition of the factor.

- the expression defining the factor is calculated and gives the component the value for the factor concerned.

- this value determines the component's category for the factor.

Example

MC

CRITERION1 = 40*VG + 40*LEVL + 20*RETU

GOOD	v2	100	50
AVERAGE	v1	v2	30
POOR	0	v1	10

CRITERION2 = 40*VG + 60*METn

GOOD	v2	100	50
AVERAGE	v1	v2	20
POOR	0	v1	0

BQ

FACTOR1: COMPONENT = CRITERION1 + CRITERION2

ACCEPTED	90	100
COMMENT	80	90
INSPECT	50	80
TEST	30	50
REWRITE	0	30

Note In this example, the sum of the highest f_weight coefficients for each quality criterion shoud be equal to 100.

Interpretation of the example

The table below gives all the possible combinations for the two criteria and the coefficient total for each combination.

CRITERION1	CRITERION2	total points	category
POOR	POOR	10 (10+0)	REWRITE
POOR	AVERAGE	30 (10+20)	TEST
POOR	GOOD	60 (10+50)	INSPECT
AVERAGE	POOR	30 (30+0)	TEST
AVERAGE	AVERAGE	50 (30+20)	INSPECT
AVERAGE	GOOD	80 (30+50)	COMMENT
GOOD	POOR	50 (50+20)	INSPECT
GOOD	AVERAGE	70 (50+20)	INSPECT
GOOD	GOOD	100 (50+50)	ACCEPTED

The total points obtained indicates the quality factor category to which the component is assigned. Value 80 in our example places the component in the quality factor's COMMENT category ([80,90[).

Example

The following example uses the possibilities of the VAL operator and the calculated value of the criterion is used instead of the weight associated to the criterion category.

MC
    CRITERION1 =   4*VG   + 4*LEVL   + 2*RETU
    ...
    ...
    CRITERION2 =   4*VG   + 6*METn
    ...
    ...
BQ
    FACTOR1 =   VAL(CRITERION1) + VAL(CRITERION2)
    VERY POOR   350   +oo
    POOR   336   350
    AVERAGE   221   336
    CORRECT   108   221
    EXCELLENT   0   108

Interpretation of the example

The table below shows how the quality factor is calculated as well as the categories associated with it.

VG	LEVL	RETU	METn	CRITERION1	CRITERION2	FACTOR	CATEGORY
0	0	0	0	0	0	0	EXCELLENT
10	3	2	2	56	52	108	CORRECT
20	7	3	5	126	95	221	AVERAGE
30	10	4	8	168	168	336	POOR
-	-	-	-	-	-	-	VERY POOR

Consider that an "excellent" component should have a VG included between 0 and 10, a LEVL between 0 and 3, an RETU between 0 and 2 and a METn between 0 and 2, the limit values carried forward to the quality factor FACTOR leads to consider as EXCELLENT the components having a value between 0 and 108.

6.2 Rule Set

This section specifies the syntax of the Logiscope Rule Set anf how it can be used to tailor Logiscope *RuleChecker* to organisation specific quality requirements.

A Rule Set is user-accessible textual file containing the specification of the programming rules to be checked by Logiscope *RuleChecker*.

Specifying one or more Rule Set files is mandatory when setting up a Logiscope *RuleChecker* project.

The Rule Sets allow to adapt Logiscope *RuleChecker* verification to a specific context taking into the applicable coding standard. See next subsection

- Rule checking can be activated or de-activated.

- Some rules have parameters that allow to customize the verification. Changing the parameters changes the behaviour of the rule checking.

- The default name of a standard rule can be changed to fit to the name and/or identifier specified in the applicable coding standard.
  The same standard rule can even be used twice with different names and different parameters.

- The default severity level of a rule can be modified.

- A new set of severity levels with a specific ordering: e.g. "Mandatory", "Highly recommended", "Recommended" can be specified.

All these actions can be done by editing the Logiscope Rule Set and changing the corresponding specification.

If you did not choose to generate a flat rule set when you created the project you may

have to edit the Rule Set file that is included in your project's Rule Set file.

Default Rule Set files are available in the **RuleSets\<language>** folder in the standard Logiscope Reference: i.e. the **Ref** folder of the Logiscope installation directory where **<language>** is the programming language of the Logiscope project under analysis: i.e. Ada, C, C++, Java.

More information on how Rule Sets are used in Logiscope projects can be found in the *IBM Rational Logiscope QualityChecker & RuleChecker Getting Started* manual.

6.2.1 Rule Set File Syntax

Using the EBNF notation, the syntax of the Rule Set file is the following :

```
<file> ::= "METRICS" <version> <line>* "END" "METRICS"
<version> ::= "VERSION" "3"
<line> ::= <severity> | <context> | <include> | <standard> |  <line_off>
<severity>::= "SEVERITY_VALUE" <level> <order> "END" "SEVERITY_VALUE"
<context> ::= "CONTEXT" <name> <status> <param>* "END" "CONTEXT"
<include> ::= "INCLUDE" <file_name> <path>* "END" "INCLUDE"
<standard> ::= "STANDARD" <name> SEVERITY <level> [RENAMING <name>]
               <status> <param>*
               "END" "STANDARD"
<line_off> ::= "CONTEXT" "OFF" | "STANDARD" "OFF"
<status> ::= "ON" | "OFF"
<level> ::= <name> | <string>
<order> ::= « an integer; example: 20»
<param> ::= {"MINMAX" <number> <number>}
               | {"LIST" <list_elem>+ "END" "LIST"}
               | <string>
<list_elem> ::= <param> | <string>
<path> ::= "IN" <path_name>
<name> ::= « sequence of letters, digits or underscore characters,
            beginning with a letter.
            Examples: asscal, Headercom, my_rule»
<file_name>::= «a sequence of letters, digits, underscore, "."
               characters.
               Example: RuleSet.rst»
<path_name>::= «a sequence of letters, digits, underscore, ".", "\"
               or "/" characters
               Example: C:\MyRuleSets\Ruleset.rst»
<number>::= «begins with an optional "-", then a sequence of digits
            and an optional decimal part ("." and a sequence of numbers)
            Examples: 1, -10, 0.345, -17.89, ...»
<string>::= « sequence of any characters surrounded with double quotes
            ("). Quotes inside the string must be doubled.
            Examples: "Highly Recommended" "no_null_stat" »
```

Comments begin with /* and end with */. They cannot be nested.

Separators are blanks, tabulations, ends of lines, and comments.

If an include line does not specify the path of the "**.rst**" file to include by using the optional "IN" part, the file will be searched for as follows:

1. in the directory of the "**.rst**" file in which the include occurs,
2. in the directories in the LOG_REF_ENV environment variable, with RuleSets*<language>* appended to them, where *<language>* is the programming language of the Logiscope project under analysis: Ada, C, C++, Java.
3. in the *<log_installation_dir>***Ref\RuleSets***<language>* directory where *<log_installation_dir>* is the Logiscope installation directory and *<language>* is the programming language of the Logiscope project under analysis: Ada, C, C++, Java.

6.2.2 Activating or De-activating a Rule Checking

Activating the checking of a given rule is just done by changing the **"OFF"** switch into **"ON"**.

For example, to check the `asscon` rule, change the associated STANDARD line in the Rule Set file as follows:

```
STANDARD asscon ON END STANDARD
```

De-activating the checking of a given rule by changing the **"ON"** switch into **"OFF"**.

6.2.3 Renaming Rules

Rule Set files with "**.rst**" format allow to rename standard rules. Such a feature is of great interest:

• to display and report rule names consistent with the user coding standard,

• to have as many versions of them as needed with different parameters.

In fact, renaming a rule can be considered as creating a new issue of an existing rule. This is done by specifying a new STANDARD line and using the RENAMING option to select the standard rule to be checked.

In the first example below, a new standard called `R04_NoAssignInCondition` is defined as a new issue of the standard rule `asscon`.

```
STANDARD R04_NoAssignInCondition RENAMING asscon ON END STANDARD
```

Note that the standard rule **asscon** may still be checked according to its status in the Rule Set. If you do not want to verify the rule anymore, you shall de-activate it (see previous section).

In the second example, 3 new rules are created renaming the same standard rule `identfmt` with different parameters ... thus giving 3 distinct naming rules ensuring consistency with the coding standard:

```
STANDARD R05_MacroNames RENAMING identfmt ON
LIST "macro"      "[A-Z0-9_x]*"
```

```
        "const"        "[A-Z0-9_x]*"
END LIST END STANDARD

STANDARD R06_TypeNames RENAMING identfmt ON
LIST "enum"          ".+_e"
     "type_struct"  ".+_t"
END LIST END STANDARD

STANDARD R07_NoDollar RENAMING identfmt ON
LIST "any"     "[^$]*"
END LIST END STANDARD
```

6.2.4 Managing Rule Severity

A level of severity is assigned to each standard rule. In most of the results provided by Logiscope *RuleChecker*, the rules are first ordered according to their level of severity.

Default Rule Sets with "**.rst**" format come with predefined levels of severity for each standard rule: i.e. "Required", "Advisory". These levels shall be considered as examples as well as the default value set for each rule.

Rule Set files allow the user to:

• specify the applicable severity levels,

• setup the severity level of each rule.

Specifying the Applicable Severity Levels

Specifying severity levels is done using the **SEVERITY_VALUE** statement in the Rule Set. The following example is extracted from the standard Rule Sets delivered with Logiscope *RuleChecker* and is compliant with the approach of the MISRA guidelines ([MISRA-C:1998] and [MISRA-C:2004]):

```
SEVERITY_VALUE Required 20 END SEVERITY_VALUE

SEVERITY_VALUE Advisory 80 END SEVERITY_VALUE
```

In the example above, two levels of severity are defined : `Required` and `Advisory`. The value following the level name is used to specify the ordering of the severity levels when displaying results. The lowest the value, the first. So 0 would refer to the highest level of severity: to be displayed first.

In the example, results related to the rules with severity `Required` are grouped and displayed first before the `Advisory` related ones:
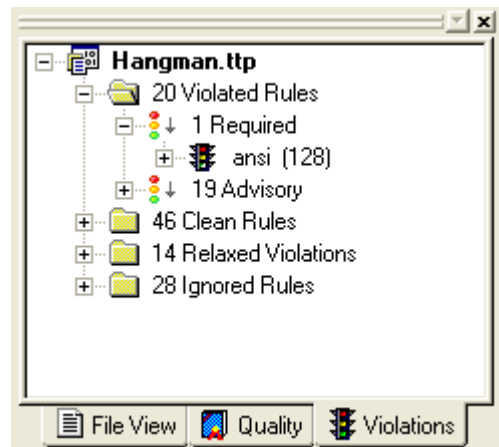
The user can easily edit the Rule Set to add more levels or even completely specify new levels according to the applicable coding standard.

The example below provides the specification of 3 levels of severity:

```
SEVERITY_VALUE Mandatory 1 END SEVERITY_VALUE

SEVERITY_VALUE "Highly Recommended" 2 END SEVERITY_VALUE

SEVERITY_VALUE Recommended 3 END SEVERITY_VALUE
```

Setting up the Severity Level for a Rule

Setting up a severity level for a rule is done using the **SEVERITY** option in the rule specification i.e. the corresponding `STANDARD` statement of the Rule Set.
This option shall be followed by one of the severity values specifed at the beginning of the Rule Set.

For instance, to change the severity level of the rule **asscon** from `Advisory` to `Required,` just modify the line:

```
STANDARD asscon SEVERITY Advisory ON END STANDARD
```

to:

```
STANDARD asscon SEVERITY Required ON END STANDARD
```

6.2.5 Importing External Violations

By checking the **Activate external violations import** box in the project settings, you can specify a project folder in which you will put files that contain violation information from external tools, for example compilation results, or the results of a program that you have written. These files will then be taken into account and their violations added to the violation information available in the project.

These data files containing the external violation information shall comply with the following format.

Import data file format

The data file to import should contain lines respecting one of the following formats:

```
"pathname" line_number "rule_mnemonic" ["message"]
```

or

```
"pathname" start_char end_char line_number "rule_mnemonic" ["message"]
```

where:

- **pathname**: is the pathname (either relative to the project, or the full path) of the source file where a violation has been found,

- **line_number**: is the line number of the violation,

- **rule_mnemonic**: is the mnemonic of the violated rule,

- **start_char**: is the position of the first character of the text to select for the violation (counting from the beginning of the file),

- **end_char**: is the position of the last character of text to select for the violation (counting from the beginning of the file),

- **message**: is an optional free text comment associated to either the violation or the rule.

When start_char and end_char are omitted, the whole line is selected when locating the violation in the file.

Each line in the file will be transformed into a violation.

Example:
```
"C:\Mastermind\machine.c" 14 "indentation"
"C:\Mastermind\machine.c" 21 "indentation"
"C:\Mastermind\machine.c" 153 160 7 "reserved_classes"
```

Files to import can have any suffix as long as the data they contain respect the specified format.

Adding a violation data file into a Logiscope Project

The files to be imported have to be added to the Logiscope Project as described below:

- Create a new folder in the Project named "Imported Files".

- Add all files to be imported in the newly created folder.
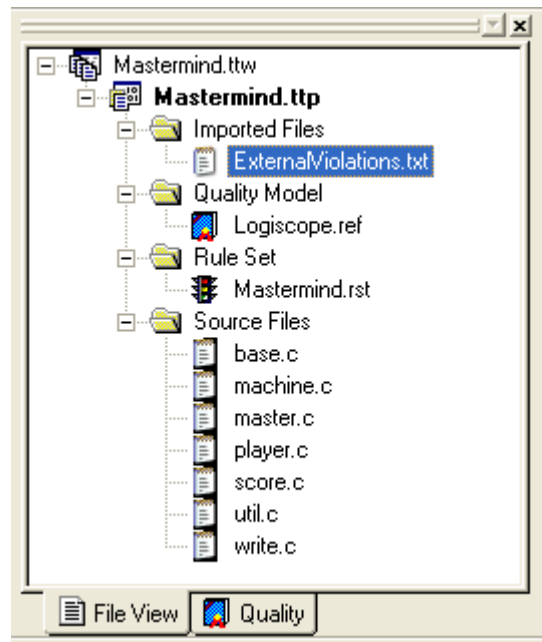
Add rule files to your project that correspond to the rules that are going to be found in the violation file. The rule files should be in the following format:

- the file for the rule rule_mnemonic should be called rule_mnemonic.std

- the contents of the file should follow this syntax:
    .NAME long_name
    .DESCRIPTION user_description
    .COMMAND external

where

- **long_name** is free text, that can include spaces. It's a more detailled title of the rule. It will appear as an explanation of the rule name in Logiscope.

- **user_description** is the description of the rule, that will be available in Logiscope.

- **external** is the type of command used for this rule, and should not be changed.

Example:

The rule file is called indentation.std.

It contains these lines:

> .NAME Use indentation
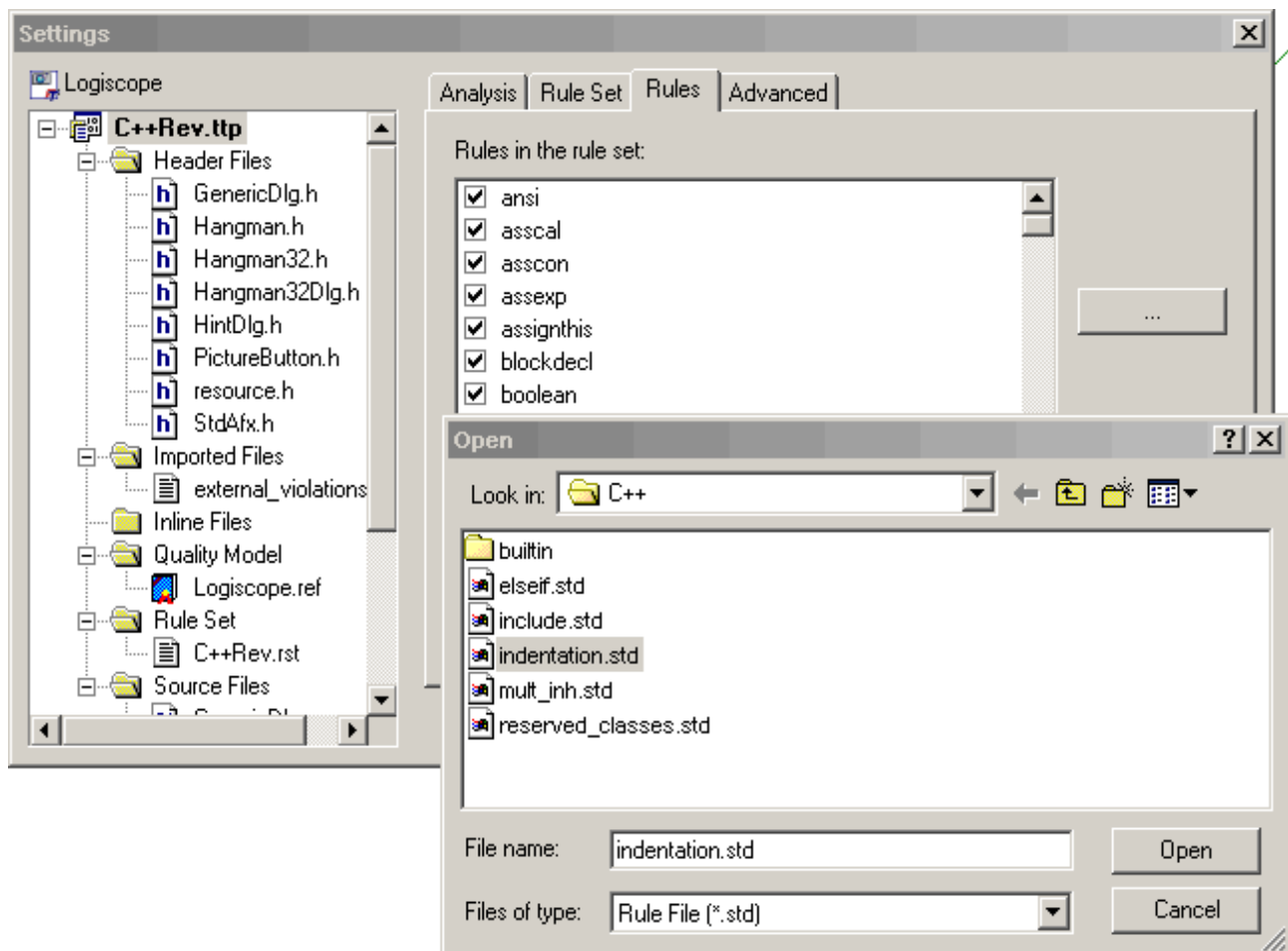> .DESCRIPTION
> Use correct indentation to clarify code.
> .COMMAND external

The external rules should be added to the project. Go to the Rules tab in the project settings. Click on the button to the right of the rule list. Select each of the rule files that you have created.

The rules should now appear in the list of project rules, and their boxes should be checked, so they are turned on.
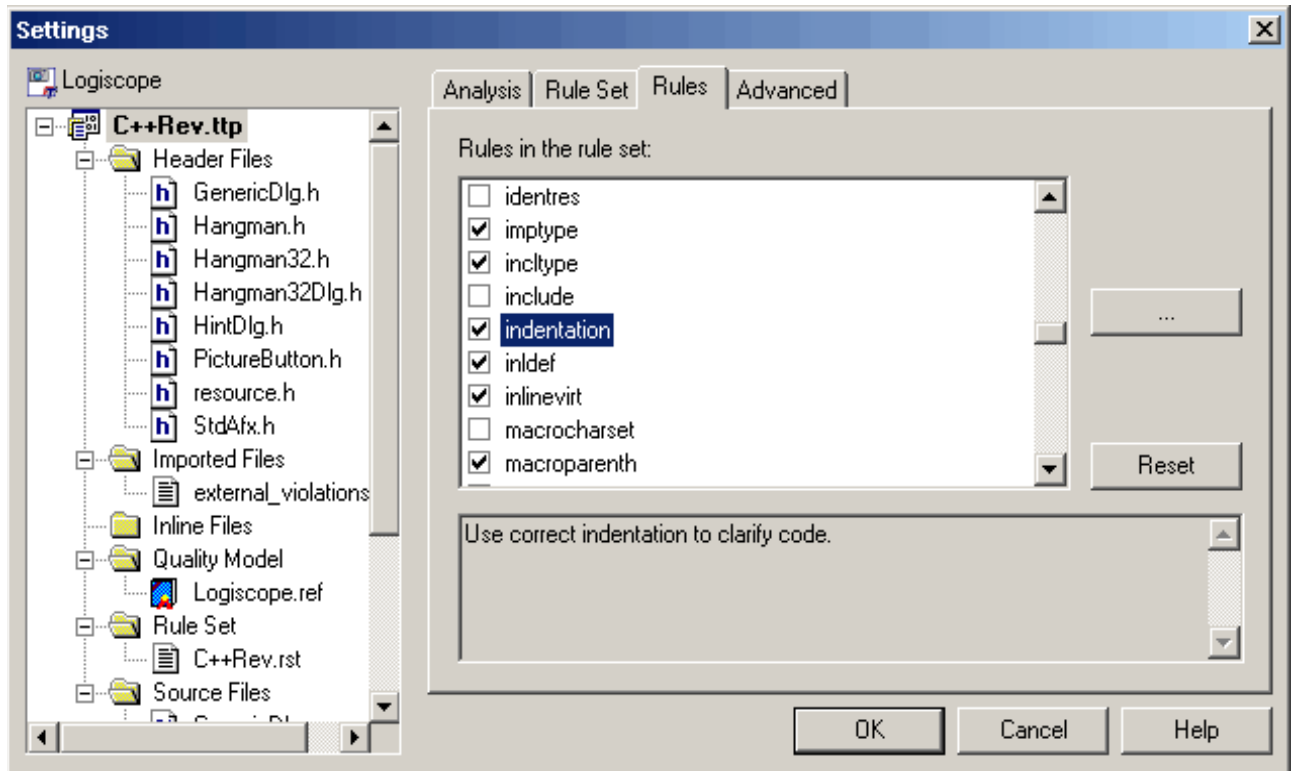
You will need to rebuild the project to see the imported violations in the Logiscope project. During the build of the project, a message such as the following should appear:

```
Loading information from cache: C:\hangman\HintDlg.h
Loading information from cache: C:\hangman\Hangman32Dlg.cpp
Loading information from cache: C:\hangman\HintDlg.cpp
Loading information from cache: C:\hangman\PictureButton.cpp
Loading information from cache: C:\hangman\StdAfx.cpp

External data import:
Importing "C:\hangman\external_violations.txt"
```
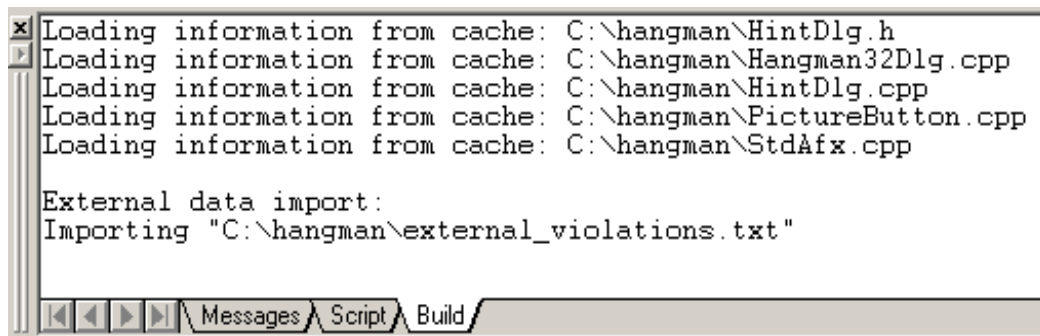
You will then see your external violations along with the Logiscope violations. In this case the violation was on a line:
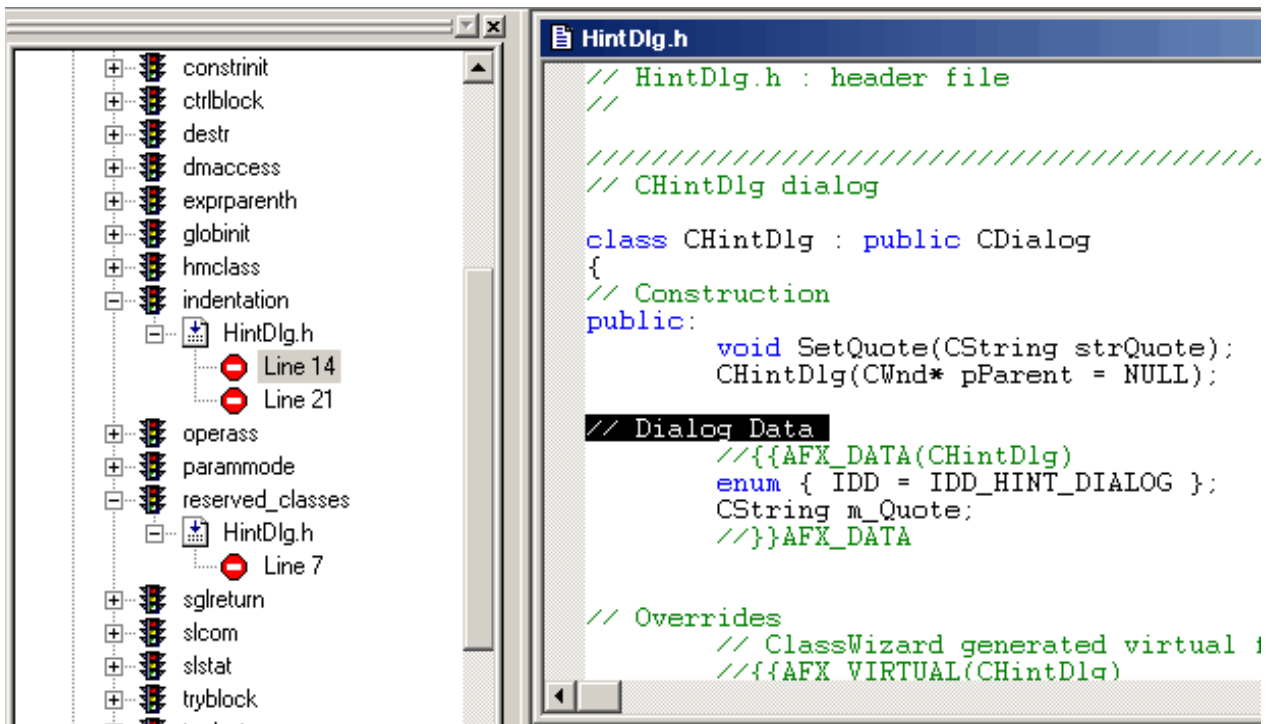
In this case, the violation used first and last character positions:
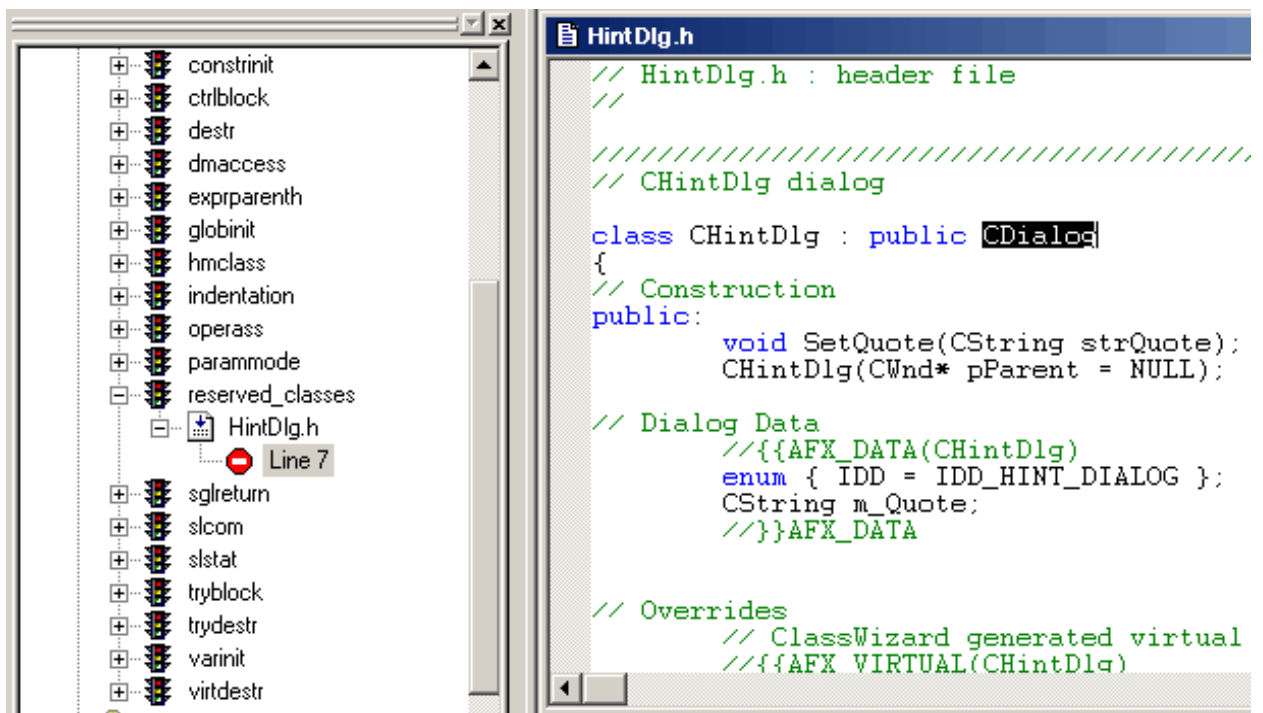
Bibliography

[2167A] DoD

Military Standard - Defense System Software Development
DOD-STD-2167A.

[610.12] IEEE

Standard Glossary of Software Engineering Terminology
IEEE 610.12-1990.

[61508-3] IEC

Functional safety of electrical / electronic / programmable electronic safety related systems - Part 3: Software requirements
IEC 61508-3:1998 (E).

[61508-7] IEC

Functional safety of electrical / electronic / programmable electronic safety related systems - Part 7: Overview on techniques and measures
IEC 61508-7:2000 (E).

[2382-1] ISO/IEC

Information technology - Vocabulary - Part 1: Fundamental terms
ISO/IEC 2382-1:1993 (E).

[8402]   ISO

Quality management and quality assurance - Vocabulary
ISO 8402:1994.

[9001]   ISO

Quality systems - Model for quality assurance in design, development, production, installation and servicing
ISO 9001:1994.

[9126-1] ISO/IEC

Software engineering - Product quality - Part 1: Quality model
ISO/IEC 9126-1:2001 (E).

[9126-2] ISO/IEC

Software engineering - Product quality - Part 2: External metrics
ISO/IEC TR 9126-2.

[9126-3] ISO/IEC

Software engineering - Product quality - Part 3: Internal metrics
ISO/IEC TR 9126-3.

[9126-4] ISO/IEC

Software engineering - Product quality - Part 4:Quality in use metrics
ISO/IEC TR 9126-4.

[12119] ISO/IEC

Information technology - Software packages - Quality requirements and testing
ISO/IEC 12119:1994 (E).

[12207] ISO/IEC

Information technology - Software lifecycle processes
ISO/IEC 12207:1995 (E).

[14598-5] ISO/IEC

Information technology - Software product evaluation - Part 5: Process for evaluators
ISO/IEC 14598-5:1998 (E).

[14598-6] ISO/IEC

Information engineering - Product evaluation - Part 6: Documentation of evaluation modules
ISO/IEC 14598-6.

[15504] ISO/IEC

Information technology - Software Process Assessment
ISO/IEC TR 15504 (all parts).

[15939] ISO/IEC

Software engineering- Software measurement process
ISO/IEC 15939:2002(E).

[DO178B] RTCA/EUROCAE

Software Considerations in Airborne Systems and Equipments Certifucation
Requirements and Technical Concepts for Aviation - RTCA SC167/DO-178B
European Organization for Civil Avaiton Electronics - EUROCAE ED-12B.

[BOEHM] B.W. BOEHM

Characteristics of Software Quality -
TRW North Holland, 1975.

Software Engineering Economics -
Prentice Hall.

[BRITO] F. BRITO E ABREU

"*Object-Oriented Software Engineering: Measuring and Controlling the Development Process*"
Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994.

"*Evaluating the Impact of Object-Oriented Design on Software Quality*"
Proceedings of the Third International Software Metrics Symposium, IEEE, Berlin, Germany, March 1996.

[GSWS]   GALILEO

Galileo Software Standard (GSWS) -
Galileo Industries - GAL-SPE-GLI-SYST-A/0092 - Issue 7, 2004.

[HAL, 77] M.H. HALSTEAD

Elements of Software Science -
North Holland, Elsevier 1977.

[HENNEL] M.A. HENNELL, D. HEDLEY, M.R. WOODWARD

Experience with Path Analysis and Testing Programs -
University of LIVERPOOL Publication.

On Program Analysis -
University of LIVERPOOL Publication.

Quantifying the Test Effectiveness of Algol 68 Programs -
University of LIVERPOOL Publication.

[McC, 76] T. McCABE

A complexity Measure -
IEEE Transaction on Software Engineering - Vol. SE-2, n.4, pp. 308-320, Dec. 1976.

[McCALL, 77] J.A. McCALL

Factors in Software Quality -
General Electric n.77C1502, June 1977.

[MEYERS-1] S. MEYERS

Effective C++: 50 Specific Ways to Improve Your Programs and Designs -
Addison-Wesley, second edition, 1997, ISBN: 0-201-92488-9.

[MEYERS-2] S. MEYERS

More Effective C++: 35 New Ways To Improve Your Programs And Designs -
Addison-Wesley, first edition, 1996, ISBN: 0-201-63371-X.

[MISRA-C:1998] MISRA

Guidelines For The Use Of The C Language In Vehicle Based Software -
Motor **I**ndustry **S**oftware **R**eliability **A**ssociation, April 1998.

[MISRA-C:2004] MISRA

MISRA-C:2004 Guidelines for the use of the C language critical systems -
Motor **I**ndustry **S**oftware **R**eliability **A**ssociation, October 2004.

[MISRA-C++:2008] MISRA

MISRA-C++:2008 Guidelines for the use of the C++ language critical systems -
Motor **I**ndustry **S**oftware **R**eliability **A**ssociation, June 2008.

[NASA] NASA IV&V,

NASA IV&V Facilities - Metrics Data Program
 http://mdp.ivv.nasa.gov/complexity_metrics.html

[NEEJ, 88] B.A. NEEJMEH

NPATH : A Measure of Execution Path Complexity and its Applications -
Communication of the ACM, 1988, Vol. 31, n.2.

[MOH, 79] S.N. MOHANTY

Models and Measurements for Quality Assessment of Software Computing Surveys -
Vol.11, n.3, September 1979.

[SIGIST] SIGIST

Glossary of terms used in Software testing -
British Computer Society - Specialist Interest Group In Software Testing

[SHT, 77] D. SCHUTT

On a Hypergraph Oriented Measure for Applied Computer Science -
Proc. COMPCON, 1977, pp. 295-296.

[WOOD, 84] M.R. WOODWARD

An Investigation into Program Paths and their Representation -
Technique et Science Informatique, 1984, Vol.8, n.4.

Terms and Definitions

Acceptance testing:
Formal testing conducted to enable a user, customer, or other authorised entity to determine whether to accept a system or component. [SIGIST]

Accuracy:
The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. [9126-1]

Acquirer:
An organisation that acquires or procures a system, software product or software service from a supplier. [12207]

Adaptability:
The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. [9126-1]

Analysability:
The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. [9126-1]

Attractiveness:
The capability of the software product to be attractive to the user. [9126-1]

Attibute:
A measurable physical or abstract property of an entity. [12207]

Black box testing:
See Functional test case design. [SIGIST]

Boundary value:
An input value or output value which is on the boundary between equivalence classes, or an incremental distance either side of the boundary.[SIGIST]

Boundary value analysis:
A test case design technique for a component in which test cases are designed which include repre-

sentatives of boundary values. [SIGRIST]

Branch:
A conditional transfer of control from any statement to any other statement in a component, or an unconditional transfer of control from any statement to any other statement in the component except the next statement, or when a component has more than one entry point, a transfer of control to an entry point of the component. [SIGIST]

Branch coverage:
The percentage of branches that have been exercised by a test case suite. [SIGIST]

Branch testing:
A test case design technique for a component in which test case are designed to execute branch outcomes. [SIGIST]

Changeability:
The capability of the software product to enable a specified modification to be implemented. [9126-1]

Code coverage:
An analysis method that determines which parts of the sofware have been executed (covered) by the test case suite and which pars have not been executed and therefore may require additional attention. [SIGIST].

Code verification:
Ensures by static verification methods the conformance of source code to the specified design of the software module, the required coding standards, and the safety planning requirements. [61508-3]

Code-based testing:
Designing tests based on objectives derived from the implementation (e.g. test that execute specific control flow paths or use specific data items. [SIGIST]

Component:
A minimal software item for which a separate specification is available. [SIGIST]

Condition:
A boolean expression containing no boolean operators. For instance A<B is a condition but A and B is not. [DO178B]

Co-existence:
The capability of the software product to co-exist with other independent software in a common environment sharing common resources. [9126-1]

Control flow:
An abstract representation of all possible sequences of events in a program's execution. [SIGIST]

Control [flow] graph:
The digrammatic representation of the possible alternative control flow paths through a component. [SIGIST]

Coverage:
The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test case suite. [SIGIST]

Coverage item:
An entity or property used as a basis for testing. [SIGIST]
Note: item can be component, statement, branch, decision, etc.

Decision:
A program point at which the control flow has two or more alternatives routes. [SIGIST]

Decision coverage:
The percentage of decision oitcomes that have been exercised by a test case suite. [SIGIST]

Developer:
An organisation that performs development activities (including requirements analysis, design, testing through acceptance) during the software lifecycle process. [12207]

Direct measure:
A measure of an attribute that does not depend upon a measure of any other attribute. [14598-1]

Dynamic analysis:
The process of evaluating a system or component based upon its behaviour during execution. [610-12]

Effectiveness:
The capability of the software product to enable users to achieve specified goals with accuracy and completeness in specified contexts of use. [9126-1]

Efficiency:
The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. [9126-1]

Entry point:
The first executable statement within a component.[SIGIST]

Error:
A human action that produces an incorrect result. [610-12]

Evaluation method:
A procedure describing the action to be performed by the evaluator in order to obtain the results for the specified measurement or verification applied on the specified product component or on the product as a whole. [14598-5]

Evaluation module:
A package of evaluation technology for a specific software quality characteristic or subcharacteristic. [14598-1]

Evaluation report:
The document that presents evaluation results and other information relevant to an evaluation. [14598-5]

Evaluation records
Documented objective evidence of all activities performed and of all results achieved within the evaluation process. [14598-5]

Evaluation requester
The person or organisation that requests an evaluation. [14598-5]

Evaluation tool
An instrument that can be used during evaluation to collect data, to perform interpretation of data or to automate part of the evaluation. [14598-5]

Evaluator
The organisation that perfoms an evaluation. [14598-5]

Executable statement:
A statement which, when compiled, is translated into object code, which will be executed procedurally when the program is running and may perform an action on program data. [SIGIST]

Exercised:
A program element is exercised by a test case when the input causes the execution of that element, such as a statement, branch, or other structural element. [SIGIST]

Exit point:
The last executable statement within a component.[SIGIST]

External measure:
An indirect measure of a product derived from measures of the behaviour of the system of which it is

a part. [14598-1]

External quality:
The extent to which a product satisfies stated and implied needs when used under specified conditions. [14598-1]

Failure:
1- Deviation of the software from its expected delivery or service. [SIGIST]
2- The termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. [14598-1]

Fault:
1- A manifestation of an error in software. A fault, if encountered may cause a failure [DO178B].
2- An incorrect step, process or data definition in a computer program [610.12]

Fault tolerance:
The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface. [9126-1]

Feasable path:
A path for which there exists a set of input values and execution conditions which causes it to be executed. [SIGIST]

Functionality:
The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. [9126-1]

Functional specification:
The document that describes in detail the characteristics of the product with regard to its intended capability. [SIGIST]

Functional test case design:
Test case selection that is based on an analysis of the specification of the component without reference to its internal workings. [SIGIST]

Indicator:
A measure that can be used to estimate or predict another measure. [14598-1]

Indirect measure:
A measure of an attribute that is derived from measures of one or more other attributes. [14598-1]

Infeasable path:
A path for which cannot be executed by any set of possible input values. [SIGIST]

Input:
A variable (whether stored within a component or outside it) that is read by the component. [SIGIST]

Inspection:
A group review quality improvment process for written material. It consists of two aspects; product (document itself) imporvement and process improvement (of both document production and inspection). [SIGIST]

Installability:
The capability of the software product to be installed in a specified environment. [9126-1]

Installability testing:
Testing concerned with the installation procedures for the system. [SIGIST]

Instrumentation:
The insertion of additional code into the program in order to collect information about program behaviour during program execution.[SIGIST]

Instrumenter:
A software tool used to carry out instrumentation.[SIGIST]

Internal measure:
A measure of the product itself, either direct or indirect. [14598-1]

Internal quality:
The totality of attributes of a product that determine its ability to satisfy stated and implied needs when used under specified conditions. [14598-1]

Interoperability:
The capability of the software product to interact with one or more specified systems. [9126-1]

Learnability:
The capability of the software product to enable the user to learn its application. [9126-1]

Level of performance:
The degree to which the needs are satisfied, represented by a specific set of values for the quality characteristics.[9126-1]

Maintainability:
Capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. [9126-1]

Maintainer:

An organisation that performs maintenance activities. [12207]

Maturity:

The capability of the software product to avoid failure as a result of faults in the software. [9126-1]

Measure (verd):

Make a measurement. [14598-1]

Measure (noun):

The number or category assigned to an attribute of an entity by making a measurement. [14598-1]

Measurement:

The use of a metric to assign a value (which may be a number or category) from a scale to an attribute of an entity. [14598-1]

Metric:

The defined measurement method and the measurement scale. [14598-1]

Modified condition/decision coverage:

The percentage of all branch condition outcomes that independenly affect a decision outcome that have been execrised by a test case suite. [DO178B]

Modified condition/decision testing:

A test case design technique in which test cases are designed to execute branch condition outcomes that independently affect a decision outcome. [DO178B]

Operability:

The capability of the software product to enable the user to operate and control it. [9126-1]

Operational testing:

Testing conducted to evaluatea system or component in its operational environment. [610.12]

Output:

A variable (whether stored within a component or outside it) that is written to by the component. [SIGIST]

Package documentation:

The product description and the user documentation.

Path:

A sequence of executable statements of a component, from an entry point to an exit point. [SIGIST]

Path coverage:
The percentage of paths in a component exercised by a test case suite. [SIGIST]

Portability:
The capability of the software product to be transferred from one environment to another. [9126-1]

Productivity:
The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use. [9126-1]

Product description:
A document stating properties of a software package, with the main purpose of helping potential buyers in the evaluation of the suitability for themselves of the product before purchasing it. [12119]

Quality:
The totality of characteristics of an entity that bear on its ability to satisfy stated or implied needs [8402].

Quality evaluation:
Systematic examination of the extent to which an entity is capable of fulfilling specified requirements [8402].

Quality in use:
The capability of the software product to enable specified users to achieve specified goals with effectiveness, productivity, safety and satisfaction in specified contexts of use. [9126-1]

Quality model:
The set of characteristics and the relationships between them which provide the basis for specifying quality requirements and evaluating quality. [14598-1]

Rating:
The action of mapping the measured value to the appropriate rating level. Used to determine the rating level associated with the software for a specific quality characteristic. [14598-1]

Rating level:
A scale point on an ordinal scale which is used to categorise a measurement scale. [14598-1]

Recoverability:
The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure. [9126-1]

Regression testing:
Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made. [SIGIST]

Reliability:
The capability of the software product to maintain a specified level of performance when used under specified conditions. [9126-1]

Replaceability:
The capability of the software product to be used in place of another specified software product for the same purpose in the same environment. [9126-1]

Requirement-based testing:
Designing tests based on objectives derived from requirements for the software component (e.g. tests that exercise specific functions or probe the non-functional constraints such as performance or security. [SIGIST]

Requirement document:
A document containing any combination of recommendations, requirements or regulations to be met by a software package. [SIGIST]

Resource utilisation:
The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions. [9126-1]

Review:
A process or meeting during which a work product, or a set of work products, is presented to project personnel, managers, users, or other interested parties for comment or approval. [610.12]

Safety:
The capability of the software product to achieve acceptable levels of risks of harm to people, business, software, property or the environment in a specified context of use. [9126-1]

Satisfaction:
The capability of the software product to satisfy users in a specified context of use. [9126-1]

Scale:
A set of value with defined properties. [14598-1]

Security:
The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them. [9126-1]

Software:
All or a part of the programs, procedures, rules and associated documentation of an information processing system. [2382-1].

Software product:
The set of computer programs, procedures, and possibly associated documentation and data. [12207]

Software product evaluation:
Technical operation that consistes of producing an assessment of one or more characteristics of a software product according to a specified procedure. [14598-5]

Stability:
The capability of the software product to avoid unexpected effects from modifications of the software. [9126-1]

Statement:
An entity in a programming language which is typically the smallest indivisible unit of execution. [SIGIST]

Statement coverage:
The percentage of executable statements in a component that have been exercised by a test case suite. [SIGIST]

Static testing:
Testing of an object without execution on a computer. [SIGIST]

Static analyser
A tool that carries out static analysis. [SIGIST]

Static analysis
Analysis of a program without execution on a computer. [SIGIST]

Structural coverage:
Coverage measures based in the internal structure of a component. [SIGIST]

Structural testing:
Test case selection that is based on an analysis of the internal structure of the component. [SIGIST]

Structured basis testing:
A test case design technique in which test cases are derived from the code logic to achieve 100% branch coverage. [SIGIST]

Structure-based testing

Designing tests based on objectives derived from the implementation (e.g., tests that execute specific control flow paths or use specific data items).

Suitability:

The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives. [9126-1]

Supplier:

An organisation that enters into a contract with the acquirer for the supply of a system, software product or software service under the terms of the contract. [12207].

System:

An integrated composite that consists or one or more of the processes, hardware, software, facilities and people, that provide a capability to satisfy a stated need of objective. [12207]

Test:

Technical operation that consists of the determination of one or more characteristics of a given product, process or service according to a specified procedure.

Testability:

The capability of the software product to enable modified software to be validated. [9126-1]

Test case:

1- A set of inputs, execution preconditions, and expected outcomes developed for a particular objective to exercise a particular program path or to verify compliance with a specific requirement. [SIGIST]

2- A documented instruction for the tester that specifies how a function or a combination of functions shall or should be tested. A test case includes detailed information on the following issues:
    (a) the test objective;
    (a) the functions to be tested;
    (a) the testing environment and other conditions;
    (a) the test data;
    (a) the procedure;
    (a) the expected behaviour of the system. [610-12]

Test case suite:

A collection of one or more test cases for the software under test. [SIGIST]

Test coverage:

See Coverage

Test plan:

A record of the test planning process detailing the degree of tester independence, the test environment, the test case design techniques and test measurement techniques to be used, and the rationale for their choice. [SIGIST]

Test procedure:
A document providing detailed instructions for the execution of one or more test cases. [SIGIST]

Test records:
For each test, an unambiguous record of the identities and versions of the component under test, the test specification, and actual outcome. [SIGIST]

Testing:
The process of exercising software to verify it satisfies requirements and to detect errors. [DO178B]

Time behaviour:
The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions. [9126-1]

Understandability:
The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. [9126-1]

Usability:
The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. [9126-1]

User:
An individual that uses the software product to perform a specific function. [14598-1]

User documentation:
The complete set of documents, available in printed or non-printed form, that is provided for the application of the product and also is an integral part of the product.

Validation:
1- Determination of the correctness of the products of software development with respect to the user needs and requirements. [SIGIST]
2- Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled. [8402]

Verification:
1- The process of evaluating a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase. [SIGIST]
2- Confirmation by examination and provision of objective evidence that specified requirements

have been fulfilled. [8402]

Walk-through:
A review of requirements, designs or code characterised by the author of the object under review guiding the progression of the review. [SIGIST]

White box testing:
See Structural testing

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, ibm.com are trademarks or registered trademarks of International Business Machine Corp., registered in many jurisdictions worldwide. Other product and services names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at:

www.ibm.com/legal/copytrade.html.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Macrovision and FLEXnet are registered trademarks or trademarks of Macrovision Corporation.

Microsoft, Windows, Windows 2003, Windows XP, Windows Vista and/or other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corporation.

Netscape and Netscape Enterprise Server are registered trademarks of Netscape Communications Corporation in the United States and other countries.

Sun, Sun Microsystems, Solaris, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Pentium is a trademark of Intel Corporation.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be trademarks or service marks of others.