M-Green, a new way for visualizing software metrics

Valentí Moncunill González
Laboratoire de recherche en génie logiciel, GÉLOG
Département de génie logiciel et des technologies de l'information
École de technologie supérieure, Montréal, QC
Email: b7743580@est.lib.upc.edu
# CONTENTS

1. **ABSTRACT** ................................................................. 4

2. **INTRODUCTION** .......................................................... 5

3. **M-GREEN’S SOFTWARE METRICS DESCRIPTION** ....................... 8
   3.1. **Introduction** .......................................................... 8
   3.2. **Descriptions** .......................................................... 8
       - Number Of Methods Overridden (NMO) ................................ 8
       - Number Of Methods Inherited (NMI) .................................... 9
       - Number Of Children (NOC) .............................................. 10
       - Number Of Methods (NOM) .............................................. 11
       - Number Of Attributes (NOA) .......................................... 12
       - Depth of Inheritance Tree (DIT) ..................................... 13
       - Abstraction (A) .......................................................... 15
       - Instability (I) .......................................................... 16
       - Specialization Index (SIX) ........................................... 18
       - Cyclomatic complexity (CC) ......................................... 20
       - Number Of Parents (NOP) ............................................. 22
       - Number of Methods Added (NMA) .................................... 23
       - Distance from the Main Sequence (DMS) ............................ 24
       - Class Category Relational Cohesion (CCRC) ......................... 25
       - Class Responsibility (CR) .......................................... 26

4. **M-GREEN DESIGN** .......................................................... 29
   - Package edu.ets.cse.green .............................................. 29
   - Package edu.ets.cse.action ............................................. 30
   - Package edu.ets.cse.constants ......................................... 31
   - Package edu.ets.csedialogs .............................................. 32
   - Package edu.ets.cse.editor ............................................. 34
   - Package edu.ets.cse.preferences ...................................... 35
   - Package edu.ets.cse.relationships .................................... 35
   - Package edu.ets.cse.types .............................................. 37
   - Package edu.ets.cse.util .............................................. 38
   - Package edu.ets.cse.views ............................................. 38
   - Package edu.ets.cse.xml ................................................. 39
   - Package edu.ets.metrics_extractor .................................. 40
   - Package edu.ets.metrics_extractor.Consts ................................ 40
   - Package edu.ets.metrics_extractor.Metrics ............................ 41
   - Package edu.ets.metrics_extractor.MetricsXMLHandler ................ 42

5. **M-GREEN IMPLEMENTATION** ............................................. 44
   - Package edu.ets.metrics_extractor .................................. 44
     Class.java ....................................................................... 44
     Main.java ....................................................................... 44
     Method.java ................................................................... 46
     MetricsContainer.java ..................................................... 46
     MetricsExtractor.java ...................................................... 46
     Package.java .................................................................. 51
     Project.java .................................................................. 52
     ProjectResource.java ...................................................... 57
6. **Using M-Green** .......................................................... 84

7. **Acknowledgment** ...................................................... 87

8. **References** ............................................................... 87
1. Abstract

Software metrics are numeric ratings used to measure the complexity and reliability of source code, the length and quality of the development process, and the performance of the application when completed.

The problem is that in the software engineering field, there is no user-friendly tool able to inform software developers if they are following the good design practices set out in the software engineering rules during the design and implementation process.

M-Green is an Eclipse plug-in developed with the objective of solving this problem since it can design UML diagrams that have the added advantage of including graphical feedback of the metrics. This lets software engineers detect typical design errors made during the design or implementation process, with the advantage that they can correct them without the whole process having to be restarted from the beginning.
2. INTRODUCTION

The motivation for developing this tool is to create an application for helping software developers during the design process of an application. The main functionality is to inform developers of typical design errors made during the design process by means of metrics values interpretations. For example, a software design with a high "coupeless" index means that classes have too much dependency between each other. This makes source code reusability and maintainability more difficult. So, it would be useful to have a tool that could inform the developer during the design process of any problems in the design whilst, at the same time, identifying the causes of these problems.

There are currently tools available that are able to calculate software metrics from a source code, but the problem is that these tools show the results of these calculations in a non user-friendly way. They sometimes give too much cryptic in numbers and this makes the task of interpreting the results more difficult for developers that are not familiar with the software metrics world. This is the problem M-Green proposes to solve by providing flexible graphical user-friendly feedback of the main software metrics both during and after the design process.

M-Green is a plug-in developed for the IDE Eclipse. The reason for integrating this tool into this development environment is because it is an IDE that is known and used by a huge number of developers. Its success is based on its open and high extensible platform. Any developer can extend its functionality by providing new plug-ins and it has recently been extended with a large number of interesting utilities.

M-Green is the result of the modification/extension of one plug-in called Green [1] and has been developed by the Research Foundation of the State University of New York. It is able to create UML diagrams using a graphic interface and then generate the source code headers of a specific programming language or simply carry out reverse engineering from source code and then generate the correspondent diagram.
The main functionality M-Green adds to this plug-in is the metrics visualization in the same UML diagrams within which developers are working.

Example of UML diagram with metrics:

M-Green is a plug-in that works together with another Eclipse plug-in called Metrics [2] that is able to calculate the main metrics values by analyzing the source code and the structure of a project.

M-Green takes advantage of this powerful tool by extracting the values of its calculated metrics. After that, M-Green can interpret them and, by means of UML diagrams, inform developers of the metrics with bad ratings.

This is an abstract scheme with the components that M-Green uses:
M-Green uses an intermediate layer called Metrics Extractor Layer for obtaining the metrics data. The purpose of putting this layer outside the M-Green is to give independence and transparency in the way M-Green accesses metrics data. This makes codes (or "the code") more reusable (other future applications will be able to use it) and maintainable since M-Green does not need to worry about future changes in Metrics plug-ins.

Also, it is possible to obtain the metrics directly from a Metrics [2] XML exported file. This has the advantage that is not necessary to have the Metrics plug-in installed as M-Green works directly with the metrics imported from the XML file. But, of course, it has the disadvantage that the metrics values shown in the UML diagrams are static. Any change in the source code or the structure of the project will not be reflected in the UML diagram.
3. **M-Green's Software Metrics Description**

3.1. Introduction

I would like to dedicate this chapter to a powerful tool called Metrics which has taken a very important part during the M-Green development. Metrics is an Eclipse Plug-in that have as an objective to compute a big set of software metrics from a certain project source code. It has implemented the algorithms which calculate the most relevant metrics used for software developers to detect errors on their software designs and implementations. In this chapter I am going to make a detailed description of the different metrics computed by Metrics and that are used at the same time by M-Green. I will put a special emphasis with the software design errors that every metric try to advice.

3.2. Descriptions

These are the software metrics computed and used by M-Green.

- Number of Methods Overridden (NMO)

1. **Overview**

The number of redefined operations plays a role in the specialization of the class and must be maintained in a proportion that continues to justify inheritance.

Too many redefined operations implies too big a difference with the parent class and inheritance then makes less sense.

2. **Computation**
For a class, this is the count of the number of inherited operations that are redefined by the class.

3. Nominal range

Between 0 and 5.

4. Analysis

A class which inherits services must use them with a minimum of modifications. If this is not the case, the inheritance loses all meaning and becomes a source of confusion.

- Number Of Methods Inherited (NMI)

1. Overview

Amongst the operations inherited by a class, the number of those which are not redefined must be relatively greater than that of those which are redefined.

2. Computation

For a class, this metric gives the percentage of the number of non-redefined operations with regard to the number of operations inherited.

\[ \text{NMI} = \frac{\text{NOHO}}{\text{HOP}} \times 100 \]

| The ... variable | represents the ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NOHO</td>
<td>number of non-redefined inherited operations</td>
</tr>
</tbody>
</table>
3. Nominal range

Between 50 % and 100 %

4. Analysis

The percentage of operations inherited should be high. This is the opposite of the Number of Methods Overridden (NMO) threshold. A low percentage of inherited operations indicates poor sub-classing.

The maximum of 100 % is ideal, but is never attained, given the fact that we often need to adapt inherited services.

- Number Of Children (NOC)

1. Overview

Inheritance, otherwise called generalization, is one of the fundamental concepts of object models, and must be used advisedly. Non-abusive use is a sign of quality and a good understanding of the concept. A class from which several classes inherit is a sensitive class, to which the user must pay great attention. It should, therefore, be limited, notably for reasons of simplicity.

2. Computation

For a class, this is the number of child classes.

For a package, this is the number of child packages.
3. Nominal range

Between 1 and 4.

4. Analysis

The upper and lower limits of 1 and 3 correspond to a desirable average. This will not stop certain particular classes being the kind of utility classes which provide services to significantly more classes than 3.

- Number Of Methods (NOM)

1. Overview

The *Number Of Methods* metric is used to calculate the average count of all class operations per class. A class must have some, but not an excessive number of operations.

This information is useful when identifying a lack of primitiveness in class operations (inhibiting re-use), and in classes which are little more than data types.

2. Computation

For a class, this is a simple count of the number of operations.

For a package, this is the average number of operations per class of the package.

3. Nominal range

Between 3 and 7.
4. Analysis

This value should remain between 3 and 7. This would indicate that a class has 
operations, but not too many. A value greater than 7 may indicate the need for further 
object-oriented decomposition, or that the class does not have a coherent purpose. A 
value of 2 or less indicates that this is not truly a class, but merely a data construction.

- Number Of Attributes (NOA)

1. Overview

The Number Of Attributes metric is used to count the average number of attributes for a 
class in the model. This information is useful in identifying the following potential 
problems:

- A class with too many attributes may indicate the presence of coincidental 
  cohesion and require further decomposition, in order to better manage the 
  complexity of the model.

- If there are no attributes, then serious attention must be paid to the semantics of 
  the class, if indeed there are any. This may be a class utility rather than a class.

2. Computation

For a class, this is a simple count of the number of attributes.

For a package, this is a count of the average number of attributes per class of the 
package.

3. Nominal range
Between 2 and 5.

4. Analysis

A high number of attributes (> 10) probably indicates poor design, notably insufficient decomposition, especially if this is associated with an equally high number of methods. Classes without attributes are particular cases, which are not necessarily anomalies. These can be interface classes, for example, which must be checked.

- Depth of Inheritance Tree (DIT)

1. Overview

Inheritance, otherwise referred to as generalization, is a key concept in the object model and must be carefully used. A class situated too deeply in the inheritance tree will be relatively complex to develop, test and maintain. It is useful, therefore, to know and regulate this depth.

This metric provides the position of the class in the inheritance tree.

2. Computation

For multiple inheritance, this metric provides the maximum length path.

- \( \text{DIT (C0)} = 0 \)
- \( \text{DIT (C0')} = 0 \)
- \( \text{DIT (C1)} = 1 \)
3. Nominal range

Between 0 and 4.

4. Analysis

A compromise between the high performance power provided by inheritance and the complexity which increases with the depth must be found. A value of between 0 and 4 respects this compromise.
A value greater than 4 would compromise encapsulation and increase complexity.

- Abstraction (A)

1. Overview

The Abstraction metric measures a package's abstraction rate. The package's abstraction level corresponds to its stability level.

Calculations are carried out on classes defined directly in the package, but also on classes defined in sub-packages or sub-classes. For a project, the metric is, therefore, calculated on all the project's classes.

The Abstraction metric provides a percentage (between 0% and 100%), where the package contains at least one class and at least one operation in an abstract class.

2. Computation

\[
Abstraction = \frac{Nma}{Nmca} \times \frac{Nca}{Nc} \times 100
\]

<table>
<thead>
<tr>
<th>The ... variable</th>
<th>represents the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nma</td>
<td>number of abstract operations in all the package's classes</td>
</tr>
<tr>
<td>Nmca</td>
<td>number of operations (abstract or not) in the package's abstract classes</td>
</tr>
<tr>
<td>Nca</td>
<td>number of abstract classes</td>
</tr>
</tbody>
</table>
The ... variable | represents the ...
--- | ---
Nc | number of classes (abstract or not) of the package

3. Nominal range

Nominal values cannot be given, since abstraction depends on what the package does (please see the "Analysis" paragraph in this section).

4. Analysis

According to how prone the package is to modification during the application's life cycle, it must be abstract to a greater or lesser extent. The more stable a package must be, the more abstract it must be, if it is to be extensible. Abstract packages that are extensible provide greater model flexibility.

This means that abstraction and instability must be jointly interpreted. This is synthesized by the Abstraction/Instability balancing metric, Distance from the Main Sequence (DMS).

- Instability (I)

1. Overview

The Instability metric measures the rate of instability of a package. A package is unstable if it depends more on other packages than they depend on it.

2. Computation
\[ I = \frac{AfferentCoupling}{EfferentCoupling + AfferentCoupling} \]

<table>
<thead>
<tr>
<th>The ... variable</th>
<th>represents the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfferentCoupling</td>
<td>number of links (associations, generalizations, use links) towards classes defined in other packages</td>
</tr>
<tr>
<td>EfferentCoupling</td>
<td>number of links (associations, generalizations, use links) coming from classes defined in other packages</td>
</tr>
</tbody>
</table>

3. **Nominal range**

Nominal values cannot be given, since instability depends on what the package does. Certain packages must be unstable whilst others must not be unstable.

4. **Analysis**

A package is that much more unstable if it depends more on other packages than they depend on it. It is likely to change if these other packages change. Each value calculated for a given package must be compared to the values of the other packages.

Not all packages have to be stable, since it must be possible for the application to evolve.

If the user wishes the package to be stable, it must depend less on the other packages than they depend on it.
- Specialization Index (SIX)

1. Overview

Redefinition and overload are undesirable because of development complexity and increased maintenance, together with the fact that they are presented at a fairly deep level in the inheritance hierarchy. To express this fact, the NMO overloading metric is multiplied by the DIT depth of inheritance. This is all related back to the total number of operations, for comparison purposes.

The metric provides a percentage, where the class contains at least one operation. For a root class, the specialization indicator is zero.

2. Computation

For a class, the specialization indicator is obtained through the following equation:

\[
SIX = \frac{NMO \times DIT}{NMI + NMA + NMO} \times 100
\]

<table>
<thead>
<tr>
<th>The ... variable</th>
<th>represents the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT</td>
<td>depth of inheritance</td>
</tr>
<tr>
<td>NMA</td>
<td>the number of operations added to the inheritance</td>
</tr>
<tr>
<td>NMI</td>
<td>the number of inherited operations</td>
</tr>
</tbody>
</table>
3. Nominal range

Between 0 % and 120 %.

4. Analysis

It is better to carry out operation redefinition as early as possible, before going more deeply into the class' inheritance graph. The more deeply we go into the inheritance, the more difficult it becomes to understand the relationship which exists between the current class and the inheritance's origin classes. Thus, redefined operations in lower levels are more difficult to develop and maintain.

The upper limit of 120 % corresponds to a number of operation re-definitions, as well as nominal operation additions to a nominal inheritance, which is NMO =3, NMA =4 and DIT =4. As for the number of non-redefined inherited operations (NMI), this is parameterized in terms of the rate with regard to the total number of inherited operations. In our case, the minimal limit for this rate is fixed at 50 %, which corresponds to a value equal to that of NMO (since NMI+NMO=100 % of inherited operations).

This is represented as follows:
\[
SLX = \frac{3 \times 4}{3 + 4 + 3} \times 100 = 120
\]

- Cyclomatic complexity (CC)

1. Overview

Cyclomatic complexity is a software metric (measurement). It was developed by Thomas McCabe and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

The concept, although not the method, is somewhat similar to that of general text complexity measured by the Flesch-Kincaid Readability Test.

Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to the commands of a program. A directed edge connects two nodes if the second command might be executed immediately after the first command.

2. Computation

\[M = E - N + 2P\]

where

- \( M \) = cyclomatic complexity
- \( E \) = the number of edges of the graph
- \( N \) = the number of nodes of the graph
- \( P \) = the number of connected components.

"\( M \)" is alternatively defined to be one larger than the number of decision points (if/else-statements, while-statements, etc) in a module (function, procedure, chart node, etc.), or more generally a system.
Separate subroutines are treated as being independent, disconnected components of the program's control flow graph.

1- Alternative definition
\[ v(G) = e - n + 2 \]

*G* is a program's flowgraph
*e* is the number of edges (arcs) in the flowgraph
*n* is the number of nodes in the flowgraph

2- Alternative way
There is another simple way to determine the cyclomatic number. This is done by counting the number of closed loops in the flow graph, and incrementing that number by one.

i.e.

\[ M = \text{Number of closed loops} + 1 \]

where

\[ M = \text{Cyclomatic number.} \]

3. **Nominal range**

   Between 0 and 15.

4. **Analysis**

   The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement containing a single condition there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.
Cyclomatic complexity is normally calculated by creating a graph of the source code with each line of source code being a node on the graph and arrows between the nodes showing the execution pathways. As some programming languages can be quite terse and compact, a source code statement when developing the graph may actually create several nodes in the graph (for instance, when using the "?:" ternary conditional operator in C, C++ and Java).

In general, in order to fully test a module all execution paths through the module should be exercised. This implies a module with a high complexity number requires more testing effort than a module with a lower value since the higher complexity number indicates more pathways through the code. This also implies that a module with higher complexity is more difficult for a programmer to understand since the programmer must understand the different pathways and the results of those pathways.

One would also expect that a module with higher complexity would tend to have lower cohesion (less than functional cohesion) than a module with lower complexity. The possible correlation between higher complexity measure with a lower level of cohesion is predicated on a module with more decision points generally implementing more than a single well defined function. However there are certain types of modules that one would expect to have a high complexity number, such as user interface (UI) modules containing source code for data validation and error recovery.

- Number Of Parents (NOP)

1. Overview

Inheritance, also known as generalization, is one of the fundamental concepts of object models and must be used advisedly. Non-abusive use is a sign of quality and of the solid understanding of the concept.

2. Computation
For a class, this is the number of parent classes.

For a package, this is the number of parent packages.

3. *Nominal range*

Between 1 and 2.

4. *Analysis*

The value 1 corresponds to a simple inheritance. Any value greater than 2 is a sign of abusive use of inheritance, unfavourable to increased simplicity.

- Number of Methods Added (NMA)

1. *Overview*

The number of operations added plays a role in the specialization of the class and must be maintained in a proportion which continues to justify inheritance, otherwise known as generalization.

Too many added operations signify too big a difference with the parent class. The inheritance would then make less sense.

2. *Computation*

For a class, this is the count of the number of operations added to the inheritance.

3. *Nominal range*
Between 0 and 4.

4. Analysis

The more added operations there are, the more the class must be redeveloped, and the less the inheritance is justified.

- Distance from the Main Sequence (DMS)

1. Overview

The Distance from the Main Sequence metric measures the balance between the abstraction and instability rates of the package.

According to what function a package has to perform, it must be able to be unstable, in other words, often significantly or abstractly modified. It must be sufficiently general to be adaptable to widely diverse situations, either without being modified or with only minimal modifications. It is preferable to have a balance between these contradictory criteria.

2. Computation

For a package, the balance between abstraction and instability is obtained through the following expression:

\[ DMS = |\text{Abstraction} + \text{Instability} - 100| \]

3. Nominal range
Between 50% and 100%.

4. Analysis

A value of 100% gives optimal balance between abstraction and instability. In practice, this optimum is never attained, and the user can be satisfied with a value greater than or equal to 50%.

- Class Category Relational Cohesion (CCRC)

1. Overview

The Class Category Relational Cohesion metric measures the rate of cohesion between a package's classes.

The grouping of classes in a package must be justified by the links that exist between its classes. The relevance of a package can be questioned, if its classes have relatively few links between themselves.

2. Computation

\[ CCRC = \frac{\text{NumberOfLinks}}{\text{NumberOfClasses}} \]

<table>
<thead>
<tr>
<th>The ... variable</th>
<th>represents the...</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberOfLinks</td>
<td>number of links (associations, generalizations, use links) between a package's classes with multiple counting if a class uses another class</td>
</tr>
<tr>
<td>The ... variable</td>
<td>represents the...</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td>in several different ways.</td>
</tr>
<tr>
<td>NumberOfClasses</td>
<td>number of classes of the package, by recursively processing sub-packages and classes. For the UML modeling project, this variable represents, therefore, the total number of classes for the UML modeling project.</td>
</tr>
</tbody>
</table>

3. *Nominal range*

   Between 150% and 350%.

4. *Analysis*

   Architecture is that much more consistent if the number of internal links in each package is relatively large. However, these links must remain within certain limits (less than 350%) for reasons of complexity.

- Class Responsibility (CR)

1. *Overview*

   The *Class Responsibility* metric provides the degree of responsibility of a class or a package.

   For a class (and respectively for a package), it gives the percentage of operations that include pre or post-conditions, with regard to the *number of operations* that the class (and respectively the package) has.
2. Computation

\[ CR = \left( \frac{PCC + POC}{2 \times NOM} \right) \times 100 \]

<table>
<thead>
<tr>
<th>The ... variable</th>
<th>represents the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCC</td>
<td>number of operations which implement pre-condition contracts</td>
</tr>
<tr>
<td>POC</td>
<td>number of operations which implement post-condition contracts</td>
</tr>
<tr>
<td>NOM</td>
<td>number of operations</td>
</tr>
</tbody>
</table>

3. Nominal range

Between 20 % and 75 %

4. Analysis

The application is more robust if classes check the conditions of use for their services and their returned results. However, too many checks can introduce a certain encumbrance which is not necessarily needed.

The fewer pre and post-conditions there are on the operations of a class, the more the class will be similar to a group of functions, rather than a consistent set of operations.

The robustness and the re-use of components which use pre and post-conditions will be increased.
Certain operations that only read attributes will not have pre and post-conditions. This explains why the upper limit of 100% is very rarely reached.

A value of less than 20% is more alarming than a value greater than 75%. 
4. **M-Green Design**

In this chapter I am going to enter in the design detail of the project. For this reason, I will describe the internal structure of M-Green using the class diagrams created while developing the project.

It is important to remark that I have divided the project in packages, so at the same time I have created one class diagram for every one of them.

- **Package edu.ets.cse.green**

  This package includes the kernel classes of the project. The main class is PlugIn and it is the core of the system. It has the interface to launch and configure properly the plug-in.

  The other important class is the GreenException. It is the class thrown as an Java exception that has the information of the possible errors that may occur during the plug-in execution.
• Package edu.ets.cse.action

This package includes the static classes that have the algorithms for extracting the metrics. The ExtractMetricsFromMetricsPluginAction simply calculate the metrics directly from the source code of the project. By the other hand, the ExtractMetricsFromXMLFileAction extract the precalculated metrics of a project stored previously on a XML file.
- **Package edu.ets.cse.constants**

It contains the constants of the plug-in. It is used only for a well organized implementation.
• Package edu.ets.cse.dialogs

This is the package that implements all the dialogs of the M-Green Plug-in.
- **Package edu.ets.cse.editor**

This package implements all the classes belonging to the class diagram editor.
- **Package edu.ets.cse.preferences**

This package contains all the menus for setting the preferences of the plug-in.

- **Package edu.ets.cse.relationships**

This package implements all different kind of relationships(association, inheritance, etc) of a UML diagram.
• Package edu.ets.cse.types

Package which defines the different types (attribute, method, etc) of a UML class.
• Package edu.ets.cse.util

This package includes some utilities, such as an image writer for writing image files.

• Package edu.ets.cse.views

It contains the class which represents the metrics container of the class figures in the UML diagram.
• Package edu.ets.cse.xml

Package with the XML parser for the MetricsExtractorFromXMLFile action.
- **Package edu.ets.metrics_extractor**

The interface for extracting the metrics from Metrics plug-in

- **Package edu.ets.metrics_extractor.Constants**

Constants for the metrics extractor.
- Package `edu.ets.metrics_extractor.Metrics`

This package contains the implementation of the different metrics used by the `metrics_extractor` interface.
- Package edu.ets.metrics_extractor.MetricsXMLHandler

Different XML file metrics extractor handlers. For now only Flat XML format files.