# Analizo
# Architecture Guide

João Miranda
Paulo Meirelles
Lucianna Almeida
Vinícius Daros
Fabio Kon
University of São Paulo (USP)

Antonio Terceiro
Joênio Costa
Luiz Romário Rios
Christina Chavez
Federal University of Bahia (UFBA)

15/06/2010

This work is partially funded by EU under the grant of IST-FP6-034763 – QualiPSo Project

http://www.qualipso.org






## Executive Summary

This document provides the technical architectural documentation of Analizo, a source code analysis toolkit.

See also Analizo user guide and installation guide.

# Table of Contents

# 1 INTRODUCTION

Analizo[1] is a multi-language source code analysis toolkit. It efficiently parses source code written in C, C++ and Java and reports useful information such as a call graph of the analized application and software metrics concerning structural complexity and size.

In order to have an easy-to-use tool which provided information for multi-language source code, Analizo was architectured to support the use of external tools as extractors. Both Analizo and each extractor are command line based applications, accepting files or directories as inputs and reporting information as output.

Analizo is written in the Perl programming language using the Object Oriented paradigm, providing both flexibility and efficiency. Doxyparse[2] is the main extractor capable of parsing C, C++ and Java (and possibly more languages) and grants great performance. It is based on Doxygen[3], a widely used open source tool for the generation of documentation through the code comments made in different programming languages.

[1] softwarelivre.org/mezuro/analizo

[2] softwarelivre.org/mezuro/doxyparse

[3] doxygen.org

# 2 ARCHITECTURE OVERVIEW

Analizo architecture is based on the use of external command line tools as source code extractors. Figure 1 presents an example of the workflow when Analizo uses Doxyparse as extractor.

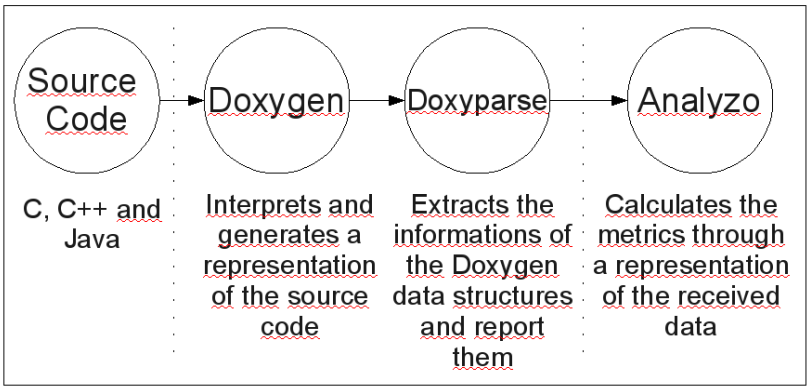


**Figure 1 – Analizo workflow using Doxyparse as extractor**

The user provides a file or directory as an input via the command line to Analizo, which executes Doxyparse. This tool is based on Doxygen internals, which parses all input files and stores a source code representation on its data structures. Afterwards, Doxyparse ends its execution by reporting information concerning modules, methods and variables found. The last step is made by Analizo which parses this output, creates a model and calculates the metrics.

## 2.1 Analizo Architecture

Analizo is mostly written in Perl, with some of its tools written in Ruby and Shell Script. Its architecture is presented in Figure 2, using a Layered style. In the diagram, each layer uses only the layers directly below it.

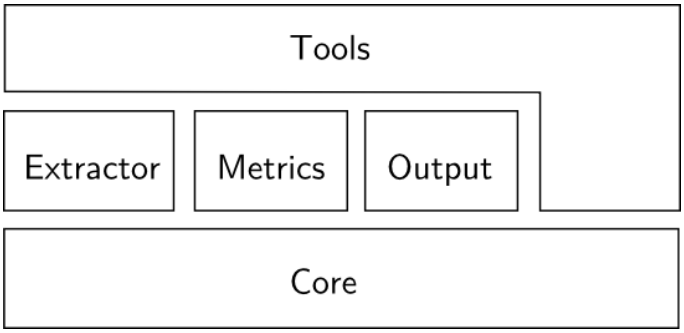


**Figure 2: Analizo's architecture, using the Layered Style**

The *Core* layer contains the data structures used to store information concerning the source code being analyzed, such as the list of existing modules \footnote{we used the ``module'' concept as a general term for the different types of structures used in software development, as classes and C source files}, elements inside each module (attributes/variables, or methods/functions), dependency information (call, inheritance etc). This layer implements most of Analizo's business logic, and it does not depend on any other layer.

The *Extractors* layer comprises the different source code information extraction strategies built in Analizo. Extractors get information from source code and store them in the *Core* layer data structures. Adding a new type of extractor that interfaces with another external tool, or provides its own analysis directly, requires only the creation of a new subclass of *Analizo::Extractor*.

There are currently two extractors, and both are interfaces for external source code parsing tools:

- Analizo::Extractors::Doxyparse is an interface for Doxyparse, a source code parser for C, C++ and Java. Doxyparse is based on Doxygen, a multi-language source code documentation system that has a robust parser.
- Analizo::Extractors::Sloccount is an interface for David A. Wheeler's Sloccount[4] a tool able to calculate the number of effective lines of code.

The other intermediate layers are *Metrics* and *Output*. The *Metrics* layer processes *Core*'s data structures in order to calculate metrics. At the moment, there is a fair set of metrics available, and they are listed in user guide documentation. The *Output* layer is designed to contain support for different

file formats. Currently, the output format implemented is the DOT format for dependency graphs, but adding new formats is a matter of adding new output handler classes.

The *Tools* layer comprises a set of command-line tools that constitute Analizo's interface for both users and higher-level applications. The tools manipulate all other layers: they instantiate the core data structures, one or more extractors, optionally the metrics processors, and an output format module, and orchestrate them in order to provide the desired result. Most of the features described in user guide documentation are implemented as Analizo tools.

Those tools are designed to adhere to the UNIX philosophy: they accomplish specialized tasks and generate output that is suitable to be fed as input to other tools, either from Analizo itself or other external tools. Some of the tools are implemented on top of others instead of explicitly manipulating Analizo's internals, and some are designed to provide output for external applications such as graph drawing programs or data analysis and visualization applications.

[4] dwheeler.com/sloccount/

### 2.1.1 Doxyparse Output

In order to truthfully understand the previous workflow, it is necessary to understand what is Doxyparse output and how Analizo uses it. The following example (Figure 3) shows the output generated using two simple Java classes (one called HelloWorld and other called Main) as inputs.

```
module HelloWorld
   function say() in line 12
      protection public
      3 lines of code
      0 parameters
      1 conditional paths
      uses variable _id defined in HelloWorld
   function HelloWorld() in line 8
      protection public
      3 lines of code
      0 parameters
      1 conditional paths
      uses variable _id defined in HelloWorld
      uses variable _id_seq defined in HelloWorld
   variable _id in line 4
   variable _id_seq in line 2
   variable hello in line 6
      protection public
module Main
   function main(String[]) in line 2
      protection public
      13 lines of code
      1 parameters
      1 conditional paths
      uses function say() defined in HelloWorld
```

**Figure 3 – Doxyparse output**

From this YAML output, we can understand that Doxyparse provides, for each module or class, a report of its methods and attributes. On the given example, the `HelloWorld` class has a method called `say()`, which is public, has 3 lines of code, 0 parameters and 1 conditional path. Notice that it also informs that `say()` uses an attribute of the current module called `_id`.

Likewise, the output shows that method `main(String[])`, defined in the `Main` class, uses this `HelloWorld` method.

### 2.1.2 How Analizo Uses Doxyparse

For each external tool used, Analizo has a corresponding extractor class responsible for the execution of the tool and the interpretation of the acquired information. The source code characteristics are stored in a Model class, which will be used, afterwards, to calculate the metrics or to generate the call graph.

With that being said, to use Doxyparse as such external tool, Analizo has a corresponding extractor class called Doxyparse, which parses the YAML and builds a Model. When a module is detected, for instance, Doxyparse extractor creates a new module representation on the Model. When it finds a function definition, it creates a function representation associated to that Module detected before.

The last step of the workflow presented in Figure 1 is the one which Analizo calculates the metrics. By the end of the interpretation of the Doxyparse output, the Model contains all the information necessary. The Metrics class just uses the Model in order to calculate and report all metrics. Finally, Figure 4 summarizes the sequence of events performed in order to parse the input and calculate its metrics.

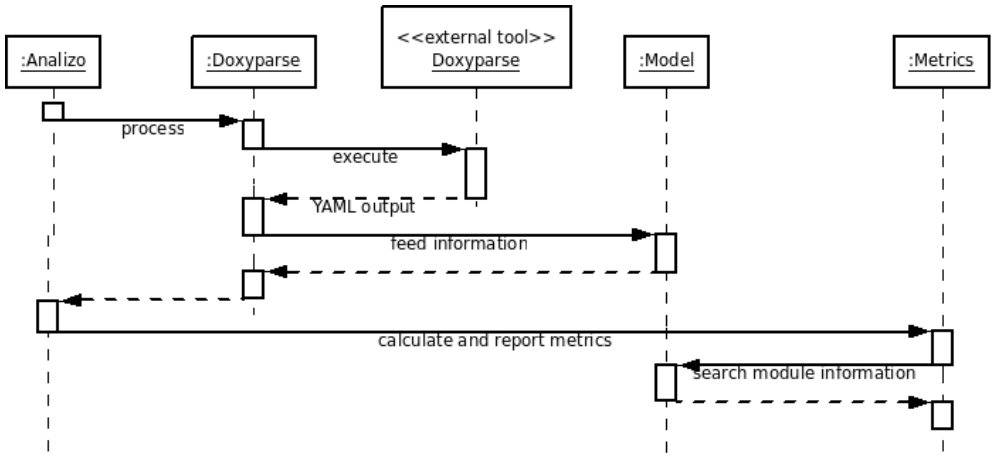


**Figure 4 – Analizo using Doxyparse to calculate metrics**

### 2.1.3 Analizo Internals

As described above, Analizo is composed by Perl scripts. The core of the toolkit are the classes that follow:

**Analizo::Model**

This class is the source code information abstraction. Therefore, it encapsulates operations to store information such as function definitions or the use of an attribute by a given module. Additionally, it provides a set of accessor methods to collect stored data.

**Analizo::Extractor**

Analizo::Extractor is the abstract superclass of all extractors. Basically, it provides an operation which instantiates any existing extractor. It also aggregates a Analizo::Model, used to store detected information. Every extractor has to define a method called `process`, responsible for the execution of the external tool and iteration over its output.

**Analizo::Metrics**

This class is responsible for all the software metrics. It communicates with clients through the `report` operation. In order to instantiate a new Analizo::Metrics, it is necessary to provide a Analizo::Model. By the time all information is stored in the Model, Analizo::Metrics iterates over all modules calculating metrics associated to each one of them and, afterwards, outputs global and per module metrics. Every metric is defined as a method of this class. For instance, `cbo` is a method of the Analizo::Metrics which calculates the Coupling between Objects metric of a given module.

**Analizo::Output::DOT**

This class is responsible for building the call graph (an example in user guide documentation). To instantiate a new Analizo::Output::DOT, it is necessary to provide a Analizo::Model that is used in the same way exposed above. The majority used operation is called `string`, which returns the output string containing the call graph. The `cluster` and the `ommit` operations enables alternative forms of the output string.

### 2.1.4 Doxyparse Internals

Doxyparse is a Doxygen branch which contains an additional Add-On called `doxyparse.cpp`. Due to that, it inherits all characteristics and data structures and uses all the source code parsers to collect the necessary information. As described before, Doxyparse runs Doxygen parsers which store information on specific data structures. Afterwards, it simply iterates over each `ClassDefinition` or `FileDefinition` reporting data of each of its members. Those members are variables/attributes and functions/methods abstracted as a `MemberDefinition`.

# 3 TROUBLESHOOTING

Analizo has no reported bugs or troubles up to date. If the user occasionally finds one, please report it on:

- http://github.com/terceiro/analizo/issues
- Mailing list:
  - mezuro@listas.softwarelivre.org
  - http://listas.softwarelivre.org/cgi-bin/mailman/listinfo/mezuro

# 4 LICENSE

# 5 ACKNOWLEDGMENT

This tool also is supported by CNPQ, FAPESB, the National Institute of Science and Technology for Software Engineering (INES), and USP FLOSS Competence Center.