SourceMeter 10.2 for JavaScript

Introduction

FrontEndART SourceMeter for JavaScript is a source code analyzer tool, which can perform deep static analysis of the source code of complex JavaScript systems (https://www.sourcemeter.com).

The source code of a program is usually its only up-to-date documentation. At the same time, the source code is the exquisite bearer of knowledge, business processes and methodology, accumulated over a long period of time. Source code quality decrease, which happens due to many quick fixes and time pressure, results in the increase of development and testing costs, and operational risks. In spite of this fact, the source code usually receives hostile treatment and is merely considered as a tool.

FrontEndART has developed SourceMeter based on the Columbus technology researched and developed at the Department of Software Engineering of the University of Szeged. SourceMeter provides deep static analysis of source code. Using the results of the analysis, the quality of the analyzed source code can be improved and developed both in the short- and long term in a directed way.

Product characteristics

The most important product characteristics of FrontEndART SourceMeter are the following:

- Platform-independent command line tools
- Transparent integration into build processes
- Powerful filter management
- Coding issue detection:
  - Metric threshold violations (MetricHunter module)
  - ESLint coding rule violations
  - TypeScript-ESLint coding rule violations in TypeScript (Experimental)
  - SONARQUBE™ platform 8.0 (“SonarQube” in the following) coding rule violations
- Clone detection (copy-pasted source code fragments) extended with clone tracking and “clone smells”
  - Syntax-based, so-called Type-2 clones
- Metrics calculation at system, file, class, method, and function levels:
  - Source code metrics
  - Clone metrics
  - Coding rule violation metrics

By continuous static analysis, the software developers can:

- reduce the software erosion rate and this way decrease development costs;
- coding problems can be identified before testing, so the number of test iterations and the testing costs can be reduced;
- the number of errors in delivered software can be reduced, so the operational risks can be decreased, increasing the company’s reputation.

SourceMeter can analyze source code conforming to ECMAScript 2022.

With the help of the filtering mechanism it is possible to specify a certain part of the ASG to be used (or not to be used) during the analysis, hence the results can be made more focused and the usage of memory and CPU can be reduced (e.g. generated source files or test code can be filtered out).

Visit our corporate website at frontendart.com to find the latest information about SourceMeter and read more about related products and services. Note, that if you are using only the free features of SourceMeter, we do not provide any free support for it. However, FrontEndART offers support services, so please feel free to search for solutions to your problems on our website and contact us with any questions you might have. We welcome feedback about the usability and eventual issues of the tool and further development suggestions at support@frontendart.com.

Background

During the static analysis, an Abstract Semantic Graph (ASG) is constructed from the language elements of the source code. This ASG is then processed by the different tools in the package to calculate product metrics, identify copy-pasted code (clones), coding rule violations, etc.

Installation

Supported platforms

SourceMeter supports the following x86 and x86-64 platforms:

- Microsoft Windows 7, 8, 8.1, and 10
- Microsoft Windows 2008 R2 Server
- Microsoft Windows 2012 Server
- GNU/Linux with kernel version 2.6.18 and GNU C library 2.11 or newer

Requirements

In order to use SourceMeter for JavaScript, it is necessary to have at least Node.js 12.x.x installed on the computer and the necessary environment variable (PATH) must be set correctly.

In case of Windows, the Microsoft Visual C++ 2017 Redistributable Package must be installed. It can be downloaded from the following URL:

- https://aka.ms/vs/15/release/vc_redist.x64.exe (x64)

The Linux package uses bash scripts.

The Linux package uses the code page conversion functions of the GNU C Library. If the conversion modules are not in the standard /usr/lib/gconv directory then the GCONV_PATH environment variable must be set according to the current installation of the GNU C Library.

E.g.: for the Windows package:

```
set "PATH=C:\Program Files\nodejs\;%PATH%"
```

E.g.: for the Linux package:

```
export "PATH=/opt/node-v10.24.1-linux-x64/bin:$PATH"
export "GCONV_PATH=/usr/lib/x86_64-linux-gnu/gconv"
```

Installation package

The SourceMeter installation package can be extracted into any folder and used from that location. It can be executed with the command line program AnalyzerJavaScript.

The SourceMeter for JavaScript package contains the following directories and files:

Windows:

Analyzer\JavaScript\		
	Demo	# Example project directory
	Tools	# ASG checker and exporter tools directory
	AnalyzerJavaScript.exe	# Program file to execute the analysis
	UsersGuide.html	# User’s guide

Linux:

Analyzer/JavaScript/		
	Demo	# Example project directory
	Tools	# ASG checker and exporter tools directory
	AnalyzerJavaScript	# Program file to execute the analysis
	UsersGuide.html	# User’s guide

License key

The free version of SourceMeter does not require any license key file.

In order to use the commercial version of SourceMeter with all features, it is necessary to obtain the license key file named LICENSE.DAT.

Windows:

The license key file must be copied into the root directory of the extracted package (next to the readme.txt file) or into the %APPDATA%\FrontEndART\SourceMeter\ directory of the user.

Linux:

The license key file must be copied into the root directory of the extracted package (next to the readme.txt file) or into the $HOME/.FrontEndART/SourceMeter/ directory of the user.

In case of the first option, anyone who has permissions to access and execute the binaries of the package will be able to use SourceMeter, while in case of the second option only those users will be able to use SourceMeter, who have the license key file in the above-mentioned directory. If one would like to renew an expired license key, the already existing license key file must be replaced with the new one in the above-mentioned directories. To use SourceMeter it is also necessary to have read and write permission on the license key file.

Command line parameters

SourceMeter can be executed with the following parameters:

-help
It displays the detailed help and exits.

-resultsDir
Relative or absolute path name of the directory where the results of the analysis will be stored. The directory will be created automatically if it does not exist.

-projectName
The name of the analyzed software system. The name specified here will be used for storing the results.

-externalHardFilter
Filter file specified with relative or absolute path, to filter out certain files from the analysis based on their path names. Filtered files will not appear in the results. The filter file is a simple text file containing lines starting with ‘+’ or ‘-’ characters followed by a regular expression[^2]. During the analysis, each input file will be checked for these expressions. If the first character of the last matching expression is ‘-’, then the given file will be excluded from the analysis. If the first character of the last matching expression is ‘+’, or there is no matching expression, then the file will be analyzed. A line starting with a different character than ‘-’ or ‘+’ will be ignored. Example filter file content:

```
# Filter out all source files starting with "test":
-test[^\.]*.js
# But test576.js is needed:
+test576.js
# Furthermore, files beginning with "test1742" are also needed:
+test1742[^\.]*.js
# Finally, test1742_b.js is not needed:
-test1742_b.js
```

-nodeOptions
Extra parameters can be added for the NodeJS used by SourceMeter. For instance the maximum heap memory of the v8 (used by Node) can be set manually if the default value is not enough.

-runMetricHunter
This parameter turns on or off the MetricHunter module. With this feature, SourceMeter lists metric threshold violations. Its value can be "true" (turn this feature on) or "false" (turn this feature off). The default value is "true".

-cloneGenealogy
This parameter turns on or off the tracking of code clones (copy-pasted source code fragments) through the consecutive revisions of the software system. It is required that during the analysis of the different revisions, the values set to projectName and resultsDir remain the same, so SourceMeter will handle them as different revisions of the same system. Its value can be "true" (turn this feature on) or "false" (turn this feature off). The default value is "false".

-cloneMinLines
This parameter sets the minimum required size of each duplication in lines of code. The default value is 20.

-csvSeparator
This parameter sets the separator character in the CSV outputs. The default value is the comma (","). The character set here must be placed in quotation marks (e.g. -csvSeparator=";"). Tabulator character can be set by the special "\t" value.

-csvDecimalMark
This parameter sets the decimal mark character in the CSV outputs. The default is value is the dot ("."). The character set here must be placed in quotation marks (e.g. -csvDecimalMark=",").

-sarifseverity
This parameter sets the severity levels to be saved in the SARIF output. (1 - Info, 2 - Minor, 3 - Major, 4 - Critical, 5 - Blocker, c/C - CloneClass). The value should not be placed in quotation marks (e.g. -sarifseverity=2345c). The default value is 2345c.

-maximumThreads
This parameter sets the maximum number of parallel tasks the controller can start. The default value is the number of available CPU cores on the current system.

-currentDate
The name of the directory with date inside the result directory of the project. If it is not set, then the current date is used.

-cleanResults
Cleans all but the last n number of timestamped result directory of the current project.

-cleanProject
Removes all files (from the directory set by the projectBaseDir parameter) created during the analysis , but does not remove anything from the results directory (resultsDir). Its value can be "true" (turn this feature on, setting projectBaseDir is mandatory in this case) or "false" (turn this feature off). The default value is "false".

-projectBaseDir
Directory of the source code to be analyzed specified with relative or absolute path. Using this option the directory analysis mode will be activated. Setting projectBaseDir is mandatory.

-runESLint
This parameter turns on or off the ESLint coding rule violation checking. With this feature, SourceMeter lists coding rule violations detected by ESLint. Its value can be "true" (turn this feature on) or "false" (turn this feature off). The default value is "true".

-runDCF
This parameter turns on or off the DuplicatedCodeFinder module. With this feature, SourceMeter identifies copy-pasted code fragments. Its value can be "true" (turn this feature on) or "false" (turn this feature off). The default value is "true".

-runMET
This parameter turns on or off the Metric module. With this feature, SourceMeter computes source code metrics. Its value can be "true" (turn this feature on) or "false" (turn this feature off). The default value is "true".

-profileXML
Global configuration file for SourceMeter. Its *tool-options* tag can be used to override the default metric thresholds for the MetricHunter tool or define custom metric formulas for the UserDefinedMetrics tool. Furthermore, its *rule-options* tag can enable/disable or modify the priorities of multiple rules. An example profile xml file is shown below:

```
        <analyzer-profile>
          <tool-options>
            <tool name = "MetricHunter" enabled = "true">
                <metric-thresholds>
                    <threshold metric-id="LOC" relation="gt" value="100" entity="Method" />
                    <threshold metric-id="LLOC" relation="gt" value="none" entity="Method" />
                </metric-thresholds>
            </tool>
          </tool-options>

          <rule-options>
            <rule id="NF" name="no-fallthrough" priority="Critical" enabled="false"/>
          </rule-options>
        </analyzer-profile>
```

-runUDM
This parameter turns on or off the UserDefinedMetrics module. With this feature, SourceMeter computes custom source code metrics based on other, previously computed metrics. Its value can be "true" (turn this feature on) or "false" (turn this feature off). The default value is dependent on the presence of custom metric definitions in the profile xml.

-runSQ
Import issues from SonarQube server.

-SQHost
The URL address of the SonarQube server.

-SQPort
The port of the SonarQube server.

-SQProjectKey
The key of the project in the SonarQube server.

-SQProjectPrefix
Prefix path of the project's base directory (the path of the sonar-project.properties file).

-SQUserName
The user name for the SonarQube server.

-SQPassword
The password for the SonarQube server.

-SQLanguageKey
The key of the language in SonarQube.

-runLIM2Patterns
This parameter can be used to enable or disable the LIM2Patterns module during analysis (default = on).

-pattern
The pattern file or pattern directory for LIM2Patterns. By default it searches for the predefined Anti Patterns found in Common/Patterns/AntiPatterns.

Usage

Execute the following command to analyse the source code of a software system:

```
SourceMeterJavaScript.exe -projectBaseDir:MyProject -projectName:MyProject -resultsDir:Results
```

Note: analyzing bundled/obfuscated source files might drastically slow down the analysis process. If the project folder contains such files (e.g. in the `dist` folder), we recommend to use the hard filter.

Result files

If any problems occur in the execution of the tools of the SourceMeter package during the code analysis, SourceMeter returns with error code 1. The error code of the given tool appears on the screen (detailed information on the bug can be found in the log directory), and the analysis is stopped, but the result files created until the failure are not deleted.

An error-free execution of SourceMeter produces the following files:

- Files containing the results of the static analysis in the results directory (set by the resultsDir parameter):
  - $(projectName)\javascript\$(DATE)\$(projectName)-*.csv

    CSV (comma separated values) files containing different metrics: Component, File, Class, Method, and Function, level source code metrics, rule violation counts, and clone-related metrics at CloneClass and CloneInstance level.
  - $(projectName)\javascript\$(DATE)\$(projectName)-clones.txt

    List of the code clones (copy-pasted source code fragments) in the system.
  - $(projectName)\javascript\$(DATE)\$(projectName)-MetricHunter.txt

    List of the MetricHunter metric value violations in the system.
  - $(projectName)\javascript\$(DATE)\$(projectName)-ESLint.txt

    List of the ESLint coding rule violations in the system.
  - $(projectName)\javascript\$(DATE)\$(projectName).graph

    Binary representation of the result graph containing all the metrics, code clones and coding rule violations.
  - $(projectName)\javascript\$(DATE)\$(projectName).xml

    XML representation of the result graph containing all the metrics, code clones and coding rule violations.
  - $(projectName)\javascript\$(projectName).gsi

    Binary data file containing information for tracking the code clones through the consecutive revisions of the analyzed software system.
  - $(projectName)\javascript\$(DATE)\$(projectName).sarif

    SARIF representation of the rule violations.
  - $(projectName)\javascript\$(DATE)\$(projectName)-summary.graph

    Binary representation of the summary result graph containing only the System component node.
  - $(projectName)\javascript\$(DATE)\$(projectName)-summary.xml

    XML representation of the summary result graph containing only the System component node.
  - $(projectName)\javascript\$(DATE)\$(projectName)-summary.json

    JSON representation of the summary result graph containing only the System component node.
- Other files and directories created in the results directory:
  - $(projectName)\javascript\$(DATE)\analyzer

    Directory, which contains configuration, binary, ASG, log, and temporary files created during the source code analysis.
  - $(projectName)\javascript\$(DATE)\analyzer\asg

    Directory, which contains backup copies of the linked ASG and LIM files, the corresponding filter files, and the GSI file.
  - $(projectName)\javascript\$(DATE)\analyzer\graph

    Directory, which contains backup copies of the graph files.
  - $(projectName)\javascript\$(DATE)\analyzer\log

    Directory, which contains the log files created during the code analysis.
  - $(projectName)\javascript\$(DATE)\analyzer\temp

    Directory, which contains the temporary files created during the code analysis.
- Files created outside the results directory:
  - .jssi

    JavaScript language dependent ASG file.

Demo

The Demo directory of the installation package contains the directory structure, the build and analyzer scripts for the analysis of an example project. The agenda open source javascript program is included as the example project, which can also be downloaded from the following URL:

https://github.com/agenda/agenda

To perform the source code analysis of the demo project, execute the analyze.bat file.

Contents of the Demo directory:

Windows:

```
Demo\
        agenda        # source code of agenda (e58fe706f4b12a8dd77768abfa8c76d4b60c5a0c)
        analyze.bat   # batch file to run the demo analysis
```

Linux:

```
Demo\
        agenda        # source code of agenda (e58fe706f4b12a8dd77768abfa8c76d4b60c5a0c)
        analyze.sh    # shell script file to run the demo analysis
```

Error messages

- Message: There is no valid license!

  Solution: Install a valid license key file (see Installation section).

- Message: Please set the resultsDir parameter. Under this directory a new directory with the name of the analyzed project will be created, if it does not exist yet. All results will be stored there under separate directories with their names containing the date and time.

  Solution: The resultsDir parameter must be set.

- Message: Please set the projectName parameter to the name of the analyzed software system.

  Solution: The projectName parameter must be set.

- Message: Please set the buildScript or projectBaseDir parameter. The usage of buildScript is recommended.

  Solution: At least one of the buildScript or projectBaseDir parameters must be set.

- Message: exec returned: 255

  If any of the log files in the the $(projectName)\javascript\$(DATE)\analyzer\log directory starts with “Could not load a transcoding service”, then the GCONV_PATH should be set to the current gconv directory. Check the “Requirements” section of the user’s guide.

- Message: exec returned: 6633

  Solution: The installed version of Node.js is lower than the required minimum (see Requirements section).

Known bugs and deficiencies

Known bugs and deficiencies of SourceMeter for JavaScripts.

- SourceMeter places the results into the directory specified by the projectName parameter. If special characters (like ‘<’, ‘>’, etc.) are used in the parameter, the analysis will probably fail.
- Variable usage edges are in experimental version. There are cases which are not handled perfectly.
- The experimental typescript parser currently used for detecting coding rule violations will return errors when it’s trying to lint files in typescript projects which aren’t part of the analyzed project (not included in either “include” or “exclude” fields in tsconfig.json). This will usually not halt the linting process, but might produce inaccurate results.

FAQ

Frequently Asked Questions regarding SourceMeter.

- Problem: After starting the analysis, neither does the computer show elevated load nor is any of the SourceMeter tools running.

  Solution: SourceMeter creates temporary files during the analysis (e.g. tmp-wrapper.bat) which might be judged as suspicious by certain antivirus programs so they may block them. In these cases, the temporary files must be manually whitelisted.

- Problem: On Windows platforms, in case of deep directory hierarchies, it may happen that the full paths of some files exceed 259 characters. In this case some SourceMeter tools may have problems with opening these files; they will terminate and write appropriate messages in the logs.

  Solution: Move the files to be analyzed to a directory closer to the file system root, or use the ‘subst’ shell command.

Appendices

Reference of source code metrics

Source code metrics are used to quantify different source code characteristics. FrontEndART SourceMeter computes source code metrics for the following source code element kinds: components, source files, classes, methods and functions.

Note that in the **free version** the following 4 metrics are not available (these metrics will get the value 0): NL, NLE, CD, TCD

The following table summarizes the metrics, their abbreviations and their correspondence to different source code element kinds:

Metric name	Abbreviation	Class	Component	File	Function	Method	Tags
API Documentation	AD	X					Documentation, SourceMeter/MET
Comment Density	CD	X			X	X	Documentation, SourceMeter/MET
Comment Lines of Code	CLOC	X		X	X	X	Documentation, SourceMeter/MET
Depth of Inheritance Tree	DIT	X					Inheritance, SourceMeter/MET
Documentation Lines of Code	DLOC	X			X	X	Documentation, SourceMeter/MET
Logical Lines of Code	LLOC	X		X	X	X	Size, SourceMeter/MET
Lines of Code	LOC	X		X	X	X	Size, SourceMeter/MET
McCabe’s Cyclomatic Complexity	McCC			X	X	X	Complexity, SourceMeter/MET
Number of Getters	NG	X					Size, SourceMeter/MET
Number of Incoming Invocations	NII	X			X	X	Coupling, SourceMeter/MET
Nesting Level	NL	X			X	X	Complexity, SourceMeter/MET
Nesting Level Else-If	NLE	X			X	X	Complexity, SourceMeter/MET
Number of Local Getters	NLG	X					Size, SourceMeter/MET
Number of Local Methods	NLM	X					Size, SourceMeter/MET
Number of Local Setters	NLS	X					Size, SourceMeter/MET
Number of Methods	NM	X					Size, SourceMeter/MET
Number of Ancestors	NOA	X					Inheritance, SourceMeter/MET
Number of Children	NOC	X					Inheritance, SourceMeter/MET
Number of Descendants	NOD	X					Inheritance, SourceMeter/MET
Number of Outgoing Invocations	NOI	X			X	X	Coupling, SourceMeter/MET
Number of Statements	NOS	X			X	X	Size, SourceMeter/MET
Number of Setters	NS	X					Size, SourceMeter/MET
Number of Parameters	NUMPAR				X	X	Size, SourceMeter/MET
Public Documented API	PDA	X					Documentation, SourceMeter/MET
Public Undocumented API	PUA	X					Documentation, SourceMeter/MET

Metric name	Abbreviation	Class	Component	File	Function	Method	Tags
Total API Documentation	TAD		X				Documentation, SourceMeter/MET
Total Comment Density	TCD	X	X		X	X	Documentation, SourceMeter/MET
Total Comment Lines of Code	TCLOC	X	X		X	X	Documentation, SourceMeter/MET
Total Logical Lines of Code	TLLOC	X	X		X	X	Size, SourceMeter/MET
Total Lines of Code	TLOC	X	X		X	X	Size, SourceMeter/MET
Total Number of Classes	TNCL		X				Size, SourceMeter/MET
Total Number of Directories	TNDI		X				Size, SourceMeter/MET
Total Number of Files	TNFI		X				Size, SourceMeter/MET
Total Number of Getters	TNG	X	X				Size, SourceMeter/MET
Total Number of Local Getters	TNLG	X					Size, SourceMeter/MET
Total Number of Local Methods	TNLM	X					Size, SourceMeter/MET
Total Number of Local Setters	TNLS	X					Size, SourceMeter/MET
Total Number of Methods	TNM	X	X				Size, SourceMeter/MET
Total Number of Statements	TNOS	X	X		X	X	Size, SourceMeter/MET
Total Number of Setters	TNS	X	X				Size, SourceMeter/MET
Total Public Documented API	TPDA		X				Documentation, SourceMeter/MET
Total Public Undocumented API	TPUA		X				Documentation, SourceMeter/MET
Weighted Methods per Class	WMC	X					Complexity, SourceMeter/MET

Definitions

API Documentation (AD)

Tags: Documentation, SourceMeter/MET

Defined: Class

Class: ratio of the number of documented methods in the class +1 if the class itself is documented to the number of all methods in the class + 1 (the class itself).

Comment Density (CD)

Tags: Documentation, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: ratio of the comment lines of the method/function (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC).

Class: ratio of the comment lines of the class (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC).

Comment Lines of Code (CLOC)

Tags: Documentation, SourceMeter/MET

Defined: Class, File, Function, Method

Method, Function: number of comment and documentation code lines of the method/function; however, its anonymous and local classes are not included.

Class: number of comment and documentation code lines of the class, including its local methods and attributes; however, its nested, anonymous, and local classes are not included.

File: number of comment and documentation code lines of the file.

Depth of Inheritance Tree (DIT)

Tags: Inheritance, SourceMeter/MET

Defined: Class

Class: length of the path that leads from the class to its farthest ancestor in the inheritance tree.

Documentation Lines of Code (DLOC)

Tags: Documentation, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: number of documentation code lines of the method/function.

Class: number of documentation code lines of the class, including its local methods and attributes; however, its nested, anonymous, and local classes are not included.

Logical Lines of Code (LLOC)

Tags: Size, SourceMeter/MET

Defined: Class, File, Function, Method

Method, Function: number of non-empty and non-comment code lines of the method/function; however, its anonymous and local functions are not included.

Class: number of non-empty and non-comment code lines of the class, including the non-empty and non-comment lines of its local methods; however, its nested, anonymous, and local classes are not included.

File: number of non-empty and non-comment code lines of the file.

Lines of Code (LOC)

Tags: Size, SourceMeter/MET

Defined: Class, File, Function, Method

Method, Function: number of code lines of the method/function, including empty and comment lines; however, its anonymous and local classes are not included.

Class: number of code lines of the class, including empty and comment lines, as well as its local methods; however, its nested, anonymous, and local classes are not included.

File: number of code lines of the file, including empty and comment lines.

McCabe's Cyclomatic Complexity (McCC)

Tags: Complexity, SourceMeter/MET

Defined: File, Function, Method

Method, Function: complexity of the method expressed as the number of independent control flow paths in it. It represents a lower bound for the number of possible execution paths in the source code and at the same time it is an upper bound for the minimum number of test cases needed for achieving full branch test coverage. The value of the metric is calculated as the number of the following instructions plus 1: if, else if, for, for…in, for…of, while, do-while, case label (which belongs to a switch instruction), catch clause, conditional expression (?:). Moreover, logical “and” (&&) and logical “or” (||) expressions also add 1 to the value because their short-circuit evaluation can cause branching depending on the first operand. The following instructions are not included: else, switch, try, finally.

File: complexity of the file expressed as the number of independent control flow paths in it. It is calculated as the sum of the McCabe's Cyclomatic Complexity values of the methods can be found in the file.

Number of Getters (NG)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of getter methods in the class, including the inherited ones.

Number of Incoming Invocations (NII)

Tags: Coupling, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: number of other methods/functions which directly call the method. If the method/function is invoked several times from the same method/function, it is counted only once.

Class: number of other methods/functions which directly call the local methods of the class. If a method is invoked several times from the same method, it is counted only once.

Nesting Level (NL)

Tags: Complexity, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: complexity of the method/function expressed as the depth of the maximum embeddedness of its conditional, iteration and exception handling block scopes. The following instructions are taken into account: if, else-if, else, for, for-in, for-of, while, do-while, switch, try, catch, finally and block statements that are directly inside another block statement. The following instructions do not increase the value by themselves; however, if additional embeddedness can be found in their blocks, they are considered: case and default label (which belong to a switch instruction).

Class: complexity of the class expressed as the depth of the maximum embeddedness of its conditional and iteration block scopes. It is calculated as the maximum nesting level (NL) of its local methods.

Nesting Level Else-If (NLE)

Tags: Complexity, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: complexity of the method/function expressed as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered. The following instructions are taken into account: if, for, while, conditional expression. The following instructions do not increase the value by themselves; however, if additional embeddedness can be found in their blocks, they are considered: else, else if (i.e. in the if-else-if construct the use of else-if does not increase the value of the metric), try, except, finally.

Class: complexity of the class expressed as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered. It is calculated as the maximum nesting level (NLE) of its local methods.

Number of Local Getters (NLG)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of local (i.e. not inherited) getter methods in the class; however, the getter methods of its nested, anonymous, and local classes are not included. Methods that override abstract methods are not counted.

Number of Local Methods (NLM)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of local (i.e. not inherited) methods in the class.

Number of Local Setters (NLS)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of local (i.e. not inherited) setter methods in the class.

Number of Methods (NM)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of methods in the class, including the inherited ones.

Number of Ancestors (NOA)

Tags: Inheritance, SourceMeter/MET

Defined: Class

Class: number of classes from which the class is directly or indirectly inherited.

Number of Children (NOC)

Tags: Inheritance, SourceMeter/MET

Defined: Class

Class: number of classes which are directly derived from the class.

Number of Descendants (NOD)

Tags: Inheritance, SourceMeter/MET

Defined: Class

Class: number of classes which are directly or indirectly derived from the class.

Number of Outgoing Invocations (NOI)

Tags: Coupling, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: number of directly called methods. If a method is invoked several times, it is counted only once.

Class: number of directly called methods of other classes, including method invocations from attribute initializations. If a method is invoked several times, it is counted only once.

Number of Statements (NOS)

Tags: Size, SourceMeter/MET

Defined: Class, Function, Method

Method, Function: number of statements in the method.

Class: number of statements in the class.

File: number of statements in the file.

Number of Setters (NS)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of setter methods in the class, including the inherited ones.

Number of Parameters (NUMPAR)

Tags: Size, SourceMeter/MET

Defined: Function, Method

Method, Function: number of the parameters of the method. Counts only the explicitly given number of parameters (‘arguments’ object is not counted).

Public Documented API (PDA)

Tags: Documentation, SourceMeter/MET

Defined: Class

Class: number of documented methods in the class (+1 if the class itself is documented); however, the methods of its nested, anonymous, and local classes are not counted.

Public Undocumented API (PUA)

Tags: Documentation, SourceMeter/MET

Defined: Class

Class: number of undocumented methods in the class (+1 if the class itself is undocumented); however, the methods of its nested, anonymous, and local classes are not counted.

Total API Documentation (TAD)

Tags: Documentation, SourceMeter/MET

Defined: Component

Component: ratio of the number of documented classes and methods in the component to the number of all of its classes and methods, including its subcomponents.

Total Comment Density (TCD)

Tags: Documentation, SourceMeter/MET

Defined: Class, Component, Function, Method

Method/Function: ratio of the total comment lines of the method/function (TCLOC) to the sum of its total comment (TCLOC) and total logical lines of code (TLLOC).

Class: ratio of the total comment lines of the class (TCLOC) to the sum of its total comment (TCLOC) and total logical lines of code (TLLOC).

Component: ratio of the total comment lines of the component (TCLOC) to the sum of its total comment (TCLOC) and total logical lines of code (TLLOC).

Total Comment Lines of Code (TCLOC)

Tags: Documentation, SourceMeter/MET

Defined: Class, Component, Function, Method

Method/Function: number of comment and documentation code lines of the method/function, including its local classes.

Class: number of comment and documentation code lines of the class, including its local methods and attributes, as well as its nested and local classes.

Component: number of comment and documentation code lines of the component, including its subcomponents.

Total Logical Lines of Code (TLLOC)

Tags: Size, SourceMeter/MET

Defined: Class, Component, Function, Method

Method, Function: number of non-empty and non-comment code lines of the method/function, including the non-empty and non-comment lines of its nested functions.

Class: number of non-empty and non-comment code lines of the class, including the non-empty and non-comment code lines of its nested and local classes.

Component: number of non-empty and non-comment code lines of the component, including its subcomponents.

Total Lines of Code (TLOC)

Tags: Size, SourceMeter/MET

Defined: Class, Component, Function, Method

Method, Function: number of code lines of the method/function, including empty and comment lines, as well as its nested functions.

Class: number of code lines of the class, including empty and comment lines, as well as its nested and local classes.

Component: number of code lines of the component, including empty and comment lines, as well as its subcomponents.

Total Number of Classes (TNCL)

Tags: Size, SourceMeter/MET

Defined: Component

Component: number of classes in the component, including the classes of its subcomponents.

Total Number of Directories (TNDI)

Tags: Size, SourceMeter/MET

Defined: Component

Component: number of directories that belong to the component, including its subcomponents.

Total Number of Files (TNFI)

Tags: Size, SourceMeter/MET

Defined: Component

Component: number of files that belong to the component, including its subcomponents.

Total Number of Getters (TNG)

Tags: Size, SourceMeter/MET

Defined: Class, Component

Class: number of getter methods in the class, including the inherited ones, as well as the inherited and local getter methods of its nested, anonymous and local classes.

Component: number of getter methods in the component, including the getter methods of its subcomponents.

Total Number of Local Getters (TNLG)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of local (i.e. not inherited) getter methods in the class, including the local getter methods of its nested, anonymous, and local classes.

Total Number of Local Methods (TNLM)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of local (i.e. not inherited) methods in the class, including the local methods of its nested and local classes.

Total Number of Local Setters (TNLS)

Tags: Size, SourceMeter/MET

Defined: Class

Class: number of local (i.e. not inherited) setter methods in the class, including the local setter methods of its nested, anonymous, and local classes.

Total Number of Methods (TNM)

Tags: Size, SourceMeter/MET

Defined: Class, Component

Class: number of methods in the class, including the inherited ones, as well as the inherited and local methods of its nested, anonymous, and local classes.

Component: number of methods in the component, including the methods of its subcomponents.

Total Number of Statements (TNOS)

Tags: Size, SourceMeter/MET

Defined: Class, Component, Function, Method

Method, Function: number of statements in the method/function, including the statements of its nested functions.Class: number of statements in the class, including the statements of its nested and local classes.Component: number of statements in the component, including the statements of its subcomponents.

Total Number of Setters (TNS)

Tags: Size, SourceMeter/MET

Defined: Class, Component

Class: number of setter methods in the class, including the inherited ones, as well as the inherited and local setter methods of its nested, anonymous and local classes.

Component: number of setter methods in the component, including the setter methods of its subcomponents.

Total Public Documented API (TPDA)

Tags: Documentation, SourceMeter/MET

Defined: Component

Component: number of documented classes and methods in the component, including the documented classes and methods of its subcomponents.

Total Public Undocumented API (TPUA)

Tags: Documentation, SourceMeter/MET

Defined: Component

Component: number of undocumented classes and methods in the component, including the undocumented classes and methods of its subcomponents.

Weighted Methods per Class (WMC)

Tags: Complexity, SourceMeter/MET

Defined: Class

Class: complexity of the class expressed as the number of independent control flow paths in it. It is calculated as the sum of the McCabe's Cyclomatic Complexity (McCC) values of its local methods.

Reference of code duplication metrics

Code cloning (or copy-paste programming) means the copying of an existing piece of source code, pasting it somewhere else, and typically performing smaller modifications on it. Based on the level of similarity between the copied code fragments we can define the following duplication types:

- **Type-1**: the copied code parts are identical code fragments except for variations in whitespace, layout and comments.
- **Type-2**: syntactically identical fragments except for variations in identifier names, literals, type references, whitespace, layout and comments.
- **Type-3**: copied fragments with further modifications such as altered, added, or removed statements, in addition to variations in identifier names, literals, type references, whitespace, layout and comments.
- **Type-4**: the copied code fragments are syntactically different, but they perform the same functionality, i.e. they are semantically the same.

FrontEndART SourceMeter is capable of identifying Type-2 clones, i.e. code fragments that are structurally identical, but may differ in variable names, literals, identifiers, etc.

Two code segments correspond to each other if they are copies of each other. This relation is an equivalence relation and we use the notion of clone classes to the classes of the relation, and the members of the classes will be referred to as clone instances. Owing to the nature of the relation, each clone class must contain at least two clone instances.

Clones are tracked during the source code analysis of consecutive revisions of the analyzed software system. FrontEndART SourceMeter detects suspicious, inconsistently changing code copies, which are called “clone smells”. The smells concerning clone classes are the following:

- **Disappearing Clone Class (DCC):** The clone class existed in the previously analyzed revision but it does not exist anymore.
- **Appearing Clone Class (ACC):** The clone class did not exist in the previously analyzed revision.

The smells concerning clone instances are issued only if the containing clone class does not contain any smell. These are the following:

- **Disappearing Clone Instance (DCI):** The clone instance existed in the previously analyzed revision but it does not exist anymore.
- **Appearing Clone Instance (ACI):** The clone instance did not exist in the previously analyzed revision.
- **Moving Clone Instance (MCI):** The clone instance belonged to a different clone class in the previously analyzed revision than its current class.

The following table summarizes the metrics, their abbreviations and their correspondence to different source code element kinds:

Metric name	Abbreviation	Class	CloneClass	CloneInstance	Component	Function	Method	Tags
Clone Age	CA		X	X				Clone, SourceMeter/DCF
Clone Coverage	CC	X			X	X	X	Clone, SourceMeter/DCF
Clone Classes	CCL	X			X	X	X	Clone, SourceMeter/DCF

Metric name	Abbreviation	Class	CloneClass	CloneInstance	Component	Function	Method	Tags
Clone Complexity	CCO	X	X	X	X	X	X	Clone, SourceMeter/DCF
Clone Embeddedness	CE		X	X				Clone, SourceMeter/DCF
Clone Elimination Effort	CEE		X		X			Clone, SourceMeter/DCF
Clone Elimination Gain	CEG		X		X			Clone, SourceMeter/DCF
Clone Instances	CI	X	X		X	X	X	Clone, SourceMeter/DCF
Clone Line Coverage	CLC	X			X	X	X	Clone, SourceMeter/DCF
Clone Logical Line Coverage	CLLC	X			X	X	X	Clone, SourceMeter/DCF
Clone Lines of Code	CLLOC		X	X				Clone, SourceMeter/DCF
Clone Risk	CR		X		X			Clone, SourceMeter/DCF
Clone Variability	CV		X	X				Clone, SourceMeter/DCF
Lines of Duplicated Code	LDC	X			X	X	X	Clone, SourceMeter/DCF
Logical Lines of Duplicated Code	LLDC	X			X	X	X	Clone, SourceMeter/DCF
Normalized Clone Radius	NCR		X		X			Clone, SourceMeter/DCF

Definitions

Clone Age (CA)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, CloneInstance

Clone class, clone instance: number of previously analyzed revisions in which the clone class/clone instance was present + 1.

Clone Coverage (CC)

Tags: Clone, SourceMeter/DCF

Defined: Class, Component, Function, Method

Method, function, class: ratio of code covered by code duplications in the source code element to the size of the source code element, expressed in terms of the number of syntactic entities (statements, expressions, etc.).

Component: ratio of code covered by code duplications in the component to the size of the component, expressed in terms of the number of syntactic entities (statements, expressions, etc.).

Clone Classes (CCL)

Tags: Clone, SourceMeter/DCF

Defined: Class, Component, Function, Method

Method, function, class: number of clone classes having at least one clone instance in the source code element.

Component: number of clone classes having at least one clone instance in the component.

Clone Complexity (CCO)

Tags: Clone, SourceMeter/DCF

Defined: Class, CloneClass, CloneInstance, Component, Function, Method

Clone instance: the McCabe complexity of the code fragment corresponding to the clone instance.

Clone class: sum of CCO of clone instances in the clone class.

Method, function, class: sum of CCO of clone instances in the source code element.

Component: sum of CCO of clone instances in the component.

Clone Embeddedness (CE)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, CloneInstance

Clone instance: sum of incoming and outgoing references (function calls, variable references, type references; different references to the same entity are counted only once) in the code fragment corresponding to the clone instance, weighted with the number of directory changes between the referenced code fragments.

Clone class: sum of CE of the clone instances of the clone class + 1.

Clone Elimination Effort (CEE)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, Component

Clone class: index of the effort required to eliminate the clone class. It is computed as the product of CI, CE, and NCR.

Component: index of the effort required to eliminate all clones from the component. It is computed as the sum of CEE of the clone classes in the component.

Clone Elimination Gain (CEG)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, Component

Clone class: index of the gain resulting from eliminating the clone class. It is computed as the ratio of CR to CEE. Component: index of the gain resulting from eliminating all clones from the component. It is computed as the logistic function of the ratio of CR to CEE.

Clone Instances (CI)

Tags: Clone, SourceMeter/DCF

Defined: Class, CloneClass, Component, Function, Method

Method, function, class: number of clone instances in the source code element.

Component: number of clone instances in the component.

Clone class: number of clone instances in the clone class.

Clone Line Coverage (CLC)

Tags: Clone, SourceMeter/DCF

Defined: Class, Component, Function, Method

Method, function, class: ratio of code covered by code duplications in the source code element to the size of the source code element, expressed in terms of lines of code.

Component: ratio of code covered by code duplications in the component to the size of the component, expressed in terms of lines of code.

Clone Logical Line Coverage (CLLC)

Tags: Clone, SourceMeter/DCF

Defined: Class, Component, Function, Method

Method, function, class: ratio of code covered by code duplications in the source code element to the size of source code element, expressed in terms of logical lines of code (non-empty, non-comment lines).

Component: ratio of code covered by code duplications in the component to the size of the component, expressed in terms of logical lines of code (non-empty, non-comment lines).

Clone Lines of Code (CLLOC)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, CloneInstance

Clone instance: length of the clone instance expressed in terms of lines of code.

Clone class: average of CLLOC of clone instances belonging to the clone class.

Clone Risk (CR)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, Component

Clone class: risk index of the existence of the clone class. It is computed as the product of CLLOC, CI, CCO, NCR, and CV.

Component: relative risk index of the existence of code duplications in the component. It is computed as the sum of CR of the clone classes in the component, divided by the total logical lines of code (non-empty, non-comment lines) of the component. It expresses the risk index projected to a non-empty, non-comment line of code in the component.

Clone Variability (CV)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, CloneInstance

Clone instance: instability of the clone instance since it appeared. It is computed as the ratio of the number of previously analyzed revisions when the instance had been changed to its age (CA).

Clone class: instability of the clone class since it appeared. It is computed as the ratio of the number of previously analyzed revisions when its contained instances were moved, deleted, or added, to its age (CA), plus the average CV of its clone instances.

Lines of Duplicated Code (LDC)

Tags: Clone, SourceMeter/DCF

Defined: Class, Component, Function, Method

Method, function, class: number of code lines covered by code duplications in the source code element.

Component: number of code lines covered by code duplications in the component.

Logical Lines of Duplicated Code (LLDC)

Tags: Clone, SourceMeter/DCF

Defined: Class, Component, Function, Method

Method, function, class: number of logical code lines (non-empty, non-comment lines) covered by code duplications in the source code element.

Component: The number of logical code lines (non-empty, non-comment lines) covered by code duplications in the component.

Normalized Clone Radius (NCR)

Tags: Clone, SourceMeter/DCF

Defined: CloneClass, Component

Clone class: normalized average distance among clone instances belonging to the clone class, expressed in terms of number of directory changes.

Component: average of NCR of the clone classes in the component.

Reference of ESLINT coding rule violations

No documentation is available.

The following table contains the enabled rules and their priorities:

Name	Abbreviation	Prio.	Tags
constructor-super	ESLINT_CSU	Major	ECMAScript 6, JavaScript, ESLINT
eqeqeq	ESLINT_E	Major	Best Practices, JavaScript, ESLINT
no-console	ESLINT_JSNC	Major	Possible Errors, JavaScript, ESLINT
no-ex-assign	ESLINT_JSNEA	Major	Possible Errors, JavaScript, ESLINT
no-cond-assign	ESLINT_NCA	Major	Possible Errors, JavaScript, ESLINT
no-class-assign	ESLINT_NCAS	Major	ECMAScript 6, JavaScript, ESLINT
no-constant-condition	ESLINT_NCC	Major	Possible Errors, JavaScript, ESLINT
no-case-declarations	ESLINT_NCD	Major	Best Practices, JavaScript, ESLINT
no-compare-neg-zero	ESLINT_NCNZ	Major	Possible Errors, JavaScript, ESLINT
no-const-assign	ESLINT_NCONS	Major	ECMAScript 6, JavaScript, ESLINT
no-control-regex	ESLINT_NCR	Major	Possible Errors, JavaScript, ESLINT
no-debugger	ESLINT_ND	Major	Possible Errors, JavaScript, ESLINT
no-dupe-args	ESLINT_NDA	Major	Possible Errors, JavaScript, ESLINT
no-duplicate-case	ESLINT_NDC	Major	Possible Errors, JavaScript, ESLINT
no-dupe-keys	ESLINT_NDK	Major	Possible Errors, JavaScript, ESLINT
no-delete-var	ESLINT_NDV	Major	Variables, JavaScript, ESLINT
no-empty	ESLINT_NE	Major	Possible Errors, JavaScript, ESLINT
no-extra-boolean-cast	ESLINT_NEBC	Major	Possible Errors, JavaScript, ESLINT
no-empty-character-class	ESLINT_NECC	Major	Possible Errors, JavaScript, ESLINT
no-empty-pattern	ESLINT_NEPA	Major	Best Practices, JavaScript, ESLINT
no-fallthrough	ESLINT_NF	Major	Best Practices, JavaScript, ESLINT
no-func-assign	ESLINT_NFA	Major	Possible Errors, JavaScript, ESLINT
no-global-assign	ESLINT_NGA	Major	Best Practices, JavaScript, ESLINT

Name	Abbreviation	Prio.	Tags
no-iterator	ESLINT_NI	Major	Best Practices, JavaScript, ESLINT
no-inner-declarations	ESLINT_NID	Major	Possible Errors, JavaScript, ESLINT
no-invalid-regexp	ESLINT_NIR	Major	Possible Errors, JavaScript, ESLINT
no-irregular-whitespace	ESLINT_NIW	Major	Possible Errors, JavaScript, ESLINT
no-mixed-spaces-and-tabs	ESLINT_NMSAT	Major	Stylistic Issues, JavaScript, ESLINT
no-new-symbol	ESLINT_NNSY	Major	ECMAScript 6, JavaScript, ESLINT
no-new-wrappers	ESLINT_NNW	Major	Best Practices, JavaScript, ESLINT
no-octal	ESLINT_NO	Major	Best Practices, JavaScript, ESLINT
no-obj-calls	ESLINT_NOC	Major	Possible Errors, JavaScript, ESLINT
no-regex-spaces	ESLINT_NRS	Major	Possible Errors, JavaScript, ESLINT
no-sparse-arrays	ESLINT_NSA	Major	Possible Errors, JavaScript, ESLINT
no-self-assign	ESLINT_NSAS	Major	Best Practices, JavaScript, ESLINT
no-this-before-super	ESLINT_NTBS	Major	ECMAScript 6, JavaScript, ESLINT
no-unreachable	ESLINT_NU	Major	Possible Errors, JavaScript, ESLINT
no-useless-escape	ESLINT_NUES	Major	Best Practices, JavaScript, ESLINT
no-unsafe-finally	ESLINT_NUF	Major	Possible Errors, JavaScript, ESLINT
no-unused-labels	ESLINT_NULA	Major	Best Practices, JavaScript, ESLINT
no-unexpected-multiline	ESLINT_NUM	Major	Possible Errors, JavaScript, ESLINT
no-undef	ESLINT_NUN	Major	Variables, JavaScript, ESLINT
no-unsafe-negation	ESLINT_NUNEG	Major	Possible Errors, JavaScript, ESLINT
require-yield	ESLINT_RY	Major	ECMAScript 6, JavaScript, ESLINT
adjacent-overload-signatures	ESLINT_TSAOS	Major	TS Stylistic Issues, TypeScript, ESLINT
array-type	ESLINT_TSAT	Major	TS Stylistic Issues, TypeScript, ESLINT
await-thenable	ESLINT_TSATH	Major	TS Best Practices, TypeScript, ESLINT
ban-types	ESLINT_TSBT	Major	TS Best Practices, TypeScript, ESLINT
ban-ts-comment	ESLINT_TSBTC	Major	TS Best Practices, TypeScript, ESLINT
no-dupe-class-members	ESLINT_TSNDCM	Major	TS Possible Errors, TypeScript, ESLINT
no-explicit-any	ESLINT_TSNEA	Major	TS Best Practices, TypeScript, ESLINT
no-empty-interface	ESLINT_TSNEI	Major	TS Possible Errors, TypeScript, ESLINT
no-extra-non-null-assertion	ESLINT_TSNENNA	Major	TS Best Practices, TypeScript, ESLINT
no-extra-parens	ESLINT_TSNEP	Major	TS Possible Errors, TypeScript, ESLINT
no-extra-semi	ESLINT_TSNES	Major	TS Possible Errors, TypeScript, ESLINT
no-for-in-array	ESLINT_TSNFIA	Major	TS Possible Errors, TypeScript, ESLINT
no-floating-promises	ESLINT_TSNFP	Major	TS Possible Errors, TypeScript, ESLINT
no-implicit-any-catch	ESLINT_TSNIAC	Major	TS Possible Errors, TypeScript, ESLINT
no-magic-numbers	ESLINT_TSNMN	Major	TS Best Practices, TypeScript, ESLINT
no-misused-promises	ESLINT_TSNMP	Major	TS Possible Errors, TypeScript, ESLINT
no-misused-new	ESLINT_TSNMW	Major	TS Variables, TypeScript, ESLINT
no-namespace	ESLINT_TSNN	Major	TS Best Practices, TypeScript, ESLINT
no-non-null-asserted-nullish-coalescing	ESLINT_TSNNNANC	Major	TS Best Practices, TypeScript, ESLINT
no-non-null-asserted-optional-chain	ESLINT_TSNNNAOC	Major	TS Possible Errors, TypeScript, ESLINT
no-redeclare	ESLINT_TSNR	Major	TS Best Practices, TypeScript, ESLINT
no-this-alias	ESLINT_TSNTA	Major	TS Variables, TypeScript, ESLINT
no-unsafe-argument	ESLINT_TSNUA	Major	TS Possible Errors, TypeScript, ESLINT
no-unsafe-assignment	ESLINT_TSNUAS	Major	TS Possible Errors, TypeScript, ESLINT
no-unsafe-call	ESLINT_TSNUCA	Major	TS Possible Errors, TypeScript, ESLINT
no-unsafe-member-access	ESLINT_TSNUMA	Major	TS Possible Errors, TypeScript, ESLINT
no-unsafe-return	ESLINT_TSNUR	Major	TS Possible Errors, TypeScript, ESLINT
no-unnecessary-type-arguments	ESLINT_TSNUTA	Major	TS Variables, TypeScript, ESLINT
no-unnecessary-type-assertion	ESLINT_TSNUTAS	Major	TS Variables, TypeScript, ESLINT
no-unnecessary-type-constraint	ESLINT_TSNUTC	Major	TS Variables, TypeScript, ESLINT
no-unused-vars	ESLINT_TSNUV	Major	TS Variables, TypeScript, ESLINT
no-var-requires	ESLINT_TSNVR	Major	TS Stylistic Issues, TypeScript, ESLINT
prefer-as-const	ESLINT_TSPAC	Major	TS Possible Errors, TypeScript, ESLINT
prefer-namespace-keyword	ESLINT_TSPNK	Major	TS Best Practices, TypeScript, ESLINT
restrict-plus-operands	ESLINT_TSRPO	Major	TS Possible Errors, TypeScript, ESLINT
triple-slash-reference	ESLINT_TSTSR	Major	TS Best Practices, TypeScript, ESLINT
unbound-method	ESLINT_TSUM	Major	TS Best Practices, TypeScript, ESLINT
unified-signatures	ESLINT_TSUS	Major	TS Stylistic Issues, TypeScript, ESLINT
use-isnan	ESLINT_UI	Major	Possible Errors, JavaScript, ESLINT
valid-typeof	ESLINT_VT	Major	Possible Errors, JavaScript, ESLINT

Definitions

constructor-super (ESLINT_CSU)

Tags: ECMAScript 6, JavaScript, ESLINT

Constructors of derived classes must call super(). Constructors of non derived classes must not call super(). If this is not observed, the javascript engine will raise a runtime error. This rule checks whether or not there is a valid super() call.

eqeqeq (ESLINT_E)

Tags: Best Practices, JavaScript, ESLINT

It is considered good practice to use the type-safe equality operators === and !== instead of their regular counterparts == and !=. The reason for this is that == and != do type coercion which follows the rather obscure Abstract Equality Comparison Algorithm. For instance, the following statements are all considered true: If one of those occurs in an innocent-looking statement such as a == b the actual problem is very difficult to spot.

no-console (ESLINT_JSNC)

Tags: Possible Errors, JavaScript, ESLINT

In JavaScript that is designed to be executed in the browser, it's considered a best practice to avoid using methods on console. Such messages are considered to be for debugging purposes and therefore not suitable to ship to the client. In general, calls using console should be stripped before being pushed to production.

no-ex-assign (ESLINT_JSNEA)

Tags: Possible Errors, JavaScript, ESLINT

If a catch clause in a try statement accidentally (or purposely) assigns another value to the exception parameter, it impossible to refer to the error from that point on. Since there is no arguments object to offer alternative access to this data, assignment of the parameter is absolutely destructive.

no-cond-assign (ESLINT_NCA)

Tags: Possible Errors, JavaScript, ESLINT

In conditional statements, it is very easy to mistype a comparison operator (such as ==) as an assignment operator (such as =). For example: There are valid reasons to use assignment operators in conditional statements. However, it can be difficult to tell whether a specific assignment was intentional.

no-class-assign (ESLINT_NCAS)

Tags: ECMAScript 6, JavaScript, ESLINT

ClassDeclaration creates a variable, and we can modify the variable. But the modification is a mistake in most cases.

no-constant-condition (ESLINT_NCC)

Tags: Possible Errors, JavaScript, ESLINT

Comparing a literal expression in a condition is usually a typo or development trigger for a specific behavior. This pattern is most likely an error and should be avoided.

no-case-declarations (ESLINT_NCD)

Tags: Best Practices, JavaScript, ESLINT

This rule disallows lexical declarations (let, const, function and class) in case/default clauses. The reason is that the lexical declaration is visible in the entire switch block but it only gets initialized when it is assigned, which will only happen if the case where it is defined is reached. To ensure that the lexical declaration only applies to the current case clause wrap your clauses in blocks.

no-compare-neg-zero (ESLINT_NCNZ)

Tags: Possible Errors, JavaScript, ESLINT

The rule should warn against code that tries to compare against -0, since that will not work as intended. That is, code like x === -0 will pass for both +0 and -0. The author probably intended Object.is(x, -0).

no-const-assign (ESLINT_NCONS)

Tags: ECMAScript 6, JavaScript, ESLINT

We cannot modify variables that are declared using const keyword. It will raise a runtime error. Under non ES2015 environment, it might be ignored merely.

no-control-regex (ESLINT_NCR)

Tags: Possible Errors, JavaScript, ESLINT

Control characters are special, invisible characters in the ASCII range 0-31. These characters are rarely used in JavaScript strings so a regular expression containing these characters is most likely a mistake.

no-debugger (ESLINT_ND)

Tags: Possible Errors, JavaScript, ESLINT

The debugger statement is used to tell the executing JavaScript environment to stop execution and start up a debugger at the current point in the code. This has fallen out of favor as a good practice with the advent of modern debugging and development tools. Production code should definitely not contain debugger, as it will cause the browser to stop executing code and open an appropriate debugger.

no-dupe-args (ESLINT_NDA)

Tags: Possible Errors, JavaScript, ESLINT

If more than one parameter has the same name in a function definition, the last occurrence “shadows” the preceding occurrences. A duplicated name might be a typing error.

no-duplicate-case (ESLINT_NDC)

Tags: Possible Errors, JavaScript, ESLINT

A switch statements with duplicate case labels is normally an indication of a programmer error. In the following example the 3rd case label uses again the literal 1 that has already been used in the first case label. Most likely the case block was copied from above and it was forgotten to change the literal.

no-dupe-keys (ESLINT_NDK)

Tags: Possible Errors, JavaScript, ESLINT

Multiple properties with the same key in object literals can cause unexpected behavior in your application.

no-delete-var (ESLINT_NDV)

Tags: Variables, JavaScript, ESLINT

The purpose of the delete operator is to remove a property from an object. Using the delete operator on a variable might lead to unexpected behavior.

no-empty (ESLINT_NE)

Tags: Possible Errors, JavaScript, ESLINT

Empty block statements, while not technically errors, usually occur due to refactoring that wasn't completed. They can cause confusion when reading code.

no-extra-boolean-cast (ESLINT_NEBC)

Tags: Possible Errors, JavaScript, ESLINT

In contexts such as an if statement's test where the result of the expression will already be coerced to a Boolean, casting to a Boolean via double negation (!!) is unnecessary. For example, these if statements are equivalent:

no-empty-character-class (ESLINT_NECC)

Tags: Possible Errors, JavaScript, ESLINT

Empty character classes in regular expressions do not match anything and can result in code that may not work as intended.

no-empty-pattern (ESLINT_NEPA)

Tags: Best Practices, JavaScript, ESLINT

When using destructuring, it's possible to create a pattern that has no effect. This happens when empty curly braces are used to the right of an embedded object destructuring pattern, such as: In this code, no new variables are created because a is just a location helper while the {} is expected to contain the variables to create, such as: In many cases, the empty object pattern is a mistake where the author intended to use a default value instead, such as: The difference between these two patterns is subtle, especially because the problematic empty pattern looks just like an object literal.

no-fallthrough (ESLINT_NF)

Tags: Best Practices, JavaScript, ESLINT

The switch statement in JavaScript is one of the more error-prone constructs of the language thanks in part to the ability to “fall through” from one case to the next. For example: In this example, if foo is 1,then execution will flow through both cases, as the first falls through to the second. You can prevent this by using break, as in this example: That works fine when you don’t want a fallthrough, but what if the fallthrough is intentional, there is no way to indicate that in the language. It’s considered a best practice to always indicate when a fallthrough is intentional using a comment: In this example, there is no confusion as to the expected behavior. It is clear that the first case is meant to fall through to the second case.

no-func-assign (ESLINT_NFA)

Tags: Possible Errors, JavaScript, ESLINT

JavaScript functions can be written as a FunctionDeclaration function foo() { … } or as a FunctionExpression var foo = function() { … };. While a JavaScript interpreter might tolerate it, overwriting/reassigning a function written as a FunctionDeclaration is often indicative of a mistake or issue.

no-global-assign (ESLINT_NGA)

Tags: Best Practices, JavaScript, ESLINT

JavaScript environments contain a number of built-in global variables, such as window in browsers and process in Node.js. In almost all cases, you don’t want to assign a value to these global variables as doing so could result in losing access to important functionality.

no-iterator (ESLINT_NI)

Tags: Best Practices, JavaScript, ESLINT

The **iterator** property was a SpiderMonkey extension to JavaScript that could be used to create custom iterators that are compatible with JavaScript’s for in and for each constructs. However, this property is now obsolete, so it should not be used. Here’s an example of how this used to work: You should use ECMAScript 6 iterators and generators instead.

no-inner-declarations (ESLINT_NID)

Tags: Possible Errors, JavaScript, ESLINT

In JavaScript, prior to ES6, a function declaration is only allowed in the first level of a program or the body of another function, though parsers sometimes erroneously accept them elsewhere. This only applies to function declarations; named or anonymous function expressions can occur anywhere an expression is permitted. A variable declaration is permitted anywhere a statement can go, even nested deeply inside other blocks. This is often undesirable due to variable hoisting, and moving declarations to the root of the program or function body can increase clarity. Note that block bindings (let, const) are not hoisted and therefore they are not affected by this rule.

no-invalid-regexp (ESLINT_NIR)

Tags: Possible Errors, JavaScript, ESLINT

An invalid pattern in a regular expression literal is a SyntaxError when the code is parsed, but an invalid string in RegExp constructors throws a SyntaxError only when the code is executed.

no-irregular-whitespace (ESLINT_NIW)

Tags: Possible Errors, JavaScript, ESLINT

Invalid or irregular whitespace causes issues with ECMAScript 5 parsers and also makes code harder to debug in a similar nature to mixed tabs and spaces. Various whitespace characters can be inputted by programmers by mistake for example from copying or keyboard shortcuts. Pressing Alt + Space on OS X adds in a non breaking space character for example. Known issues these spaces cause:

no-mixed-spaces-and-tabs (ESLINT_NMSAT)

Tags: Stylistic Issues, JavaScript, ESLINT

Most code conventions require either tabs or spaces be used for indentation. As such, it’s usually an error if a single line of code is indented with both tabs and spaces.

no-new-symbol (ESLINT_NNSY)

Tags: ECMAScript 6, JavaScript, ESLINT

Using a single import statement per module will make the code clearer because you can see everything being imported from that module on one line. In the following example the module import on line 1 is repeated on line 3. These can be combined to make the list of imports more succinct.

no-new-wrappers (ESLINT_NNW)

Tags: Best Practices, JavaScript, ESLINT

There are three primitive types in JavaScript that have wrapper objects: string, number, and boolean. These are represented by the constructors String, Number, and Boolean, respectively. The primitive wrapper types are used whenever one of these primitive values is read, providing them with object-like capabilities such as methods. Behind the scenes, an object of the associated wrapper type is created and then destroyed, which is why you can call methods on primitive values, such as: Behind the scenes in this example, a String object is constructed. The substring() method exists on String.prototype and so is accessible to the string instance. It’s also possible to manually create a new wrapper instance: Although possible, there aren’t any good reasons to use these primitive wrappers as constructors. They tend to confuse other developers more than anything else because they seem like they should act as primitives, but they do not. For example: The first problem is that primitive wrapper objects are, in fact, objects. That means typeof will return “object” instead of “string”, “number”, or “boolean”. The second problem comes with boolean objects. Every object is truthy, that means an instance of Boolean always resolves to true even when its actual value is false. For these reasons, it’s considered a best practice to avoid using primitive wrapper types with new.

no-octal (ESLINT_NO)

Tags: Best Practices, JavaScript, ESLINT

Octal literals are numerals that begin with a leading zero, such as: The leading zero to identify an octal literal has been a source of confusion and error in JavaScript. ECMAScript 5 deprecates the use of octal numeric literals in JavaScript and octal literals cause syntax errors in strict mode. It’s therefore recommended to avoid using octal literals in JavaScript code.

no-obj-calls (ESLINT_NOC)

Tags: Possible Errors, JavaScript, ESLINT

ECMAScript provides several global objects that are intended to be used as-is. Some of these objects look as if they could be constructors due their capitalization (such as Math and JSON) but will throw an error if you try to execute them as functions. The ECMAScript 5 specification makes it clear that both Math and JSON cannot be invoked:

no-regex-spaces (ESLINT_NRS)

Tags: Possible Errors, JavaScript, ESLINT

Regular expressions can be very complex and difficult to understand, which is why it’s important to keep them as simple as possible in order to avoid mistakes. One of the more error-prone things you can do with a regular expression is to use more than one space, such as: In this regular expression, it’s very hard to tell how many spaces are intended to be matched. It’s better to use only one space and then specify how many spaces are expected, such as: Now it is very clear that three spaces are expected to be matched.

no-sparse-arrays (ESLINT_NSA)

Tags: Possible Errors, JavaScript, ESLINT

Sparse arrays contain empty slots, most frequently due to multiple commas being used in an array literal, such as: While the items array in this example has a length of 2, there are actually no values in items[0] or items[1]. The fact that the array literal is valid with only commas inside, coupled with the length being set and actual item values not being set, make sparse arrays confusing for many developers. Consider the following: In this example, the colors array has a length of 3. But did the developer intend for there to be an empty spot in the middle of the array? Or is it a typo? The confusion around sparse arrays defined in this manner is enough that it’s recommended to avoid using them unless you are certain that they are useful in your code.

no-self-assign (ESLINT_NSAS)

Tags: Best Practices, JavaScript, ESLINT

Self assignments have no effect, so probably those are an error due to incomplete refactoring. Those indicate that what you should do is still remaining.

no-this-before-super (ESLINT_NTBS)

Tags: ECMAScript 6, JavaScript, ESLINT

In the constructor of derived classes, if this/super are used before super() calls, it raises a reference error. This rule checks this/super keywords in constructors, then reports those that are before super().

no-unreachable (ESLINT_NU)

Tags: Possible Errors, JavaScript, ESLINT

Because the return, throw, break, and continue statements unconditionally exit a block of code, any statements after them cannot be executed. Unreachable statements are usually a mistake.

no-useless-escape (ESLINT_NUES)

Tags: Best Practices, JavaScript, ESLINT

Escaping non-special characters in strings, template literals, and regular expressions doesn't have any effect, as demonstrated in the following example:

no-unsafe-finally (ESLINT_NUF)

Tags: Possible Errors, JavaScript, ESLINT

JavaScript suspends the control flow statements of try and catch blocks until the execution of finally block finishes. So, when return, throw, break, or continue is used in finally, control flow statements inside try and catch are overwritten, which is considered as unexpected behavior.

no-unused-labels (ESLINT_NULA)

Tags: Best Practices, JavaScript, ESLINT

The –fix option on the command line can automatically fix some of the problems reported by this rule. Labels that are declared and not used anywhere in the code are most likely an error due to incomplete refactoring.

no-unexpected-multiline (ESLINT_NUM)

Tags: Possible Errors, JavaScript, ESLINT

Semicolons are optional in JavaScript, via a process called automatic semicolon insertion (ASI). See the documentation for semi for a fuller discussion of that feature. The rules for ASI are relatively straightforward: In short, as once described by Isaac Schlueter, a character always ends a statement (just like a semicolon) unless one of the following is true: This particular rule aims to spot scenarios where a newline looks like it is ending a statement, but is not.

no-undef (ESLINT_NUN)

Tags: Variables, JavaScript, ESLINT

This rule can help you locate potential ReferenceErrors resulting from misspellings of variable and parameter names, or accidental implicit globals (for example, from forgetting the var keyword in a for loop initializer).

no-unsafe-negation (ESLINT_NUNEG)

Tags: Possible Errors, JavaScript, ESLINT

Just as developers might type -a + b when they mean -(a + b) for the negative of a sum, they might type !key in object by mistake when they almost certainly mean !(key in object) to test that a key is not in an object. !obj instanceof Ctor is similar.

require-yield (ESLINT_RY)

Tags: ECMAScript 6, JavaScript, ESLINT

This rule generates warnings for generator functions that do not have the yield keyword.

adjacent-overload-signatures (ESLINT_TSAOS)

Tags: TS Stylistic Issues, TypeScript, ESLINT

This rule aims to standardize the way overloaded members are organized.

array-type (ESLINT_TSAT)

Tags: TS Stylistic Issues, TypeScript, ESLINT

This rule aims to standardize usage of array types within your codebase.

await-thenable (ESLINT_TSATH)

Tags: TS Best Practices, TypeScript, ESLINT

-

ban-types (ESLINT_TSBT)

Tags: TS Best Practices, TypeScript, ESLINT

This rule bans specific types and can suggest alternatives. Note that it does not ban the corresponding runtime objects from being used.

ban-ts-comment (ESLINT_TSBTC)

Tags: TS Best Practices, TypeScript, ESLINT

This rule lets you set which directive comments you want to allow in your codebase. By default, only @ts-check is allowed, as it enables rather than suppresses errors.

no-dupe-class-members (ESLINT_TSNDCM)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule extends the base eslint/no-dupe-class-members rule. It adds support for TypeScript's method overload definitions.

no-explicit-any (ESLINT_TSNEA)

Tags: TS Best Practices, TypeScript, ESLINT

This rule doesn't allow any types to be defined. It aims to keep TypeScript maximally useful. TypeScript has a compiler flag for –noImplicitAny that will prevent an any type from being implied by the compiler, but doesn't prevent any from being explicitly used.

no-empty-interface (ESLINT_TSNEI)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule aims to ensure that only meaningful interfaces are declared in the code.

no-extra-non-null-assertion (ESLINT_TSNENNA)

Tags: TS Best Practices, TypeScript, ESLINT

Disallow extra non-null assertion.

no-extra-parens (ESLINT_TSNEP)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule extends the base eslint/no-extra-parens rule. It adds support for TypeScript type assertions.

no-extra-semi (ESLINT_TSNES)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule extends the base eslint/no-extra-semi rule. It adds support for class properties.

no-for-in-array (ESLINT_TSNFIA)

Tags: TS Possible Errors, TypeScript, ESLINT

A for-in loop (for (var k in o)) iterates over the properties of an Object. While it is legal to use for-in loops with array types, it is not common. for-in will iterate over the indices of the array as strings, omitting any "holes" in the array. More common is to use for-of, which iterates over the values of an array.

no-floating-promises (ESLINT_TSNFP)

Tags: TS Possible Errors, TypeScript, ESLINT

Requires Promise-like values to be handled appropriately. This rule forbids usage of Promise-like values in statements without handling their errors appropriately. Unhandled promises can cause several issues, such as improperly sequenced operations, ignored Promise rejections and more. Valid ways of handling a Promise-valued statement include awaiting, returning, and either calling .then() with two arguments or .catch() with one argument.

no-implicit-any-catch (ESLINT_TSNIAC)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule requires an explicit type to be declared on a catch clause variable.

no-magic-numbers (ESLINT_TSNMN)

Tags: TS Best Practices, TypeScript, ESLINT

This rule extends the base eslint/no-magic-numbers rule. It adds support for: numeric literal types, enum members, and readonly class properties.

no-misused-promises (ESLINT_TSNMP)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule forbids using promises in places where the TypeScript compiler allows them but they are not handled properly. These situations can often arise due to a missing await keyword or just a misunderstanding of the way async functions are handled/awaited.

no-misused-new (ESLINT_TSNMW)

Tags: TS Variables, TypeScript, ESLINT

Warns on apparent attempts to define constructors for interfaces or new for classes.

no-namespace (ESLINT_TSNN)

Tags: TS Best Practices, TypeScript, ESLINT

This rule aims to standardize the way modules are declared.

no-non-null-asserted-nullish-coalescing (ESLINT_TSNNNANC)

Tags: TS Best Practices, TypeScript, ESLINT

The nullish coalescing operator is designed to provide a default value when dealing with null or undefined. Using non-null assertions in the left operand of the nullish coalescing operator is redundant.

no-non-null-asserted-optional-chain (ESLINT_TSNNNAOC)

Tags: TS Possible Errors, TypeScript, ESLINT

Optional chain expressions are designed to return undefined if the optional property is nullish. Using non-null assertions after an optional chain expression is wrong, and introduces a serious type safety hole into your code.

no-redeclare (ESLINT_TSNR)

Tags: TS Best Practices, TypeScript, ESLINT

This rule extends the base eslint/no-redeclare rule. It adds support for TypeScript function overloads, and declaration merging.

no-this-alias (ESLINT_TSNTA)

Tags: TS Variables, TypeScript, ESLINT

This rule prohibits assigning variables to this.

no-unsafe-argument (ESLINT_TSNUA)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule disallows calling a function with any in its arguments, and it will disallow spreading any[]. This rule also disallows spreading a tuple type with one of its elements typed as any. This rule also compares the argument's type to the variable's type to ensure you don't pass an unsafe any in a generic position to a receiver that's expecting a specific type. For example, it will error if you assign Set any to an argument declared as Set string.

no-unsafe-assignment (ESLINT_TSNUAS)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule disallows assigning any to a variable, and assigning any[] to an array destructuring. This rule also compares the assigned type to the variable's type to ensure you don't assign an unsafe any in a generic position to a receiver that's expecting a specific type. For example, it will error if you assign Set any to a variable declared as Set string.

no-unsafe-call (ESLINT_TSNUCA)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule disallows calling any variable that is typed as any.

no-unsafe-member-access (ESLINT_TSNUMA)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule disallows member access on any variable that is typed as any.

no-unsafe-return (ESLINT_TSNUR)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule disallows returning any or any[] from a function. This rule also compares the return type to the function's declared/inferred return type to ensure you don't return an unsafe any in a generic position to a receiver that's expecting a specific type. For example, it will error if you return Set any from a function declared as returning Set string.

no-unnecessary-type-arguments (ESLINT_TSNUTA)

Tags: TS Variables, TypeScript, ESLINT

Type parameters in TypeScript may specify a default value.

no-unnecessary-type-assertion (ESLINT_TSNUTAS)

Tags: TS Variables, TypeScript, ESLINT

This rule aims to prevent unnecessary type assertions.

no-unnecessary-type-constraint (ESLINT_TSNUTC)

Tags: TS Variables, TypeScript, ESLINT

-

no-unused-vars (ESLINT_TSNUV)

Tags: TS Variables, TypeScript, ESLINT

This rule extends the base eslint/no-unused-vars rule. It adds support for TypeScript features, such as types.

no-var-requires (ESLINT_TSNVR)

Tags: TS Stylistic Issues, TypeScript, ESLINT

-

prefer-as-const (ESLINT_TSPAC)

Tags: TS Possible Errors, TypeScript, ESLINT

This rule recommends usage of const assertion when type primitive value is equal to type.

prefer-namespace-keyword (ESLINT_TSPNK)

Tags: TS Best Practices, TypeScript, ESLINT

This rule aims to standardize the way modules are declared.

restrict-plus-operands (ESLINT_TSRPO)

Tags: TS Possible Errors, TypeScript, ESLINT

-

triple-slash-reference (ESLINT_TSTSR)

Tags: TS Best Practices, TypeScript, ESLINT

-

unbound-method (ESLINT_TSUM)

Tags: TS Best Practices, TypeScript, ESLINT

Warns when a method is used outside of a method call.

unified-signatures (ESLINT_TSUS)

Tags: TS Stylistic Issues, TypeScript, ESLINT

This rule aims to keep the source code as maintainable as possible by reducing the amount of overloads.

use-isnan (ESLINT_UI)

Tags: Possible Errors, JavaScript, ESLINT

In JavaScript, NaN is a special value of the Number type. It's used to represent any of the “not-a-number” values represented by the double-precision 64-bit format as specified by the IEEE Standard for Binary Floating-Point Arithmetic. NaN has the unique property of not being equal to anything, including itself. That is to say, that the condition NaN !== NaN evaluates to true.

valid-typeof (ESLINT_VT)

Tags: Possible Errors, JavaScript, ESLINT

For a vast majority of use-cases, the only valid results of the typeof operator will be one of the following: “undefined”, “object”, “boolean”, “number”, “string”, and “function”. When the result of a typeof operation is compared against a string that is not one of these strings, it is usually a typo. This rule ensures that when the result of a typeof operation is compared against a string, that string is in the aforementioned set.

Tool Tags

No documentation is available.

Name	Rules
ESLINT	ESLINT
SourceMeter	SourceMeter
SourceMeter/DCF	CA, CC, CCL, CCO, CE, CEE, CEG, CI, CLC, CLLC, CLLOC, CR, CV, LDC, LLDC, NCR
SourceMeter/MET	AD, CD, CLOC, DIT, DLOC, LLOC, LOC, McCC, NG, NII, NL, NLE, NLG, NLM, NLS, NM, NOA, NOC, NOD, NOI, NOS, NS, NUMPAR, PDA, PUA, TAD, TCD, TCLOC, TLLOC, TLOC, TNCL, TNDI, TNFI, TNG, TNLG, TNLM, TNLS, TNM, TNOS, TNS, TPDA, TPUA, WMC

General Tags

No documentation is available.

Name	Rules
Best Practices	ESLINT_E, ESLINT_NCD, ESLINT_NEPA, ESLINT_NF, ESLINT_NGA, ESLINT_NI, ESLINT_NNW, ESLINT_NO, ESLINT_NSAS, ESLINT_NUES, ESLINT_NULA
Clone	CA, CC, CCL, CCO, CE, CEE, CEG, CI, CLC, CLLC, CLLOC, CR, CV, LDC, LLDC, NCR
Complexity	McCC, NL, NLE, WMC
Coupling	NII, NOI
Documentation	AD, CD, CLOC, DLOC, PDA, PUA, TAD, TCD, TCLOC, TPDA, TPUA
ECMAScript 6	ESLINT_CSU, ESLINT_NCAS, ESLINT_NCONS, ESLINT_NNSY, ESLINT_NTBS, ESLINT_RY
Inheritance	DIT, NOA, NOC, NOD
Possible Errors	ESLINT_JSNC, ESLINT_JSNEA, ESLINT_NCA, ESLINT_NCC, ESLINT_NCNZ, ESLINT_NCR, ESLINT_ND, ESLINT_NDA, ESLINT_NDC, ESLINT_NDK, ESLINT_NE, ESLINT_NEBC, ESLINT_NECC, ESLINT_NFA, ESLINT_NID, ESLINT_NIR, ESLINT_NIW, ESLINT_NOC, ESLINT_NRS, ESLINT_NSA, ESLINT_NU, ESLINT_NUF, ESLINT_NUM, ESLINT_NUNEG, ESLINT_UI, ESLINT_VT
Size	LLOC, LOC, NG, NLG, NLM, NLS, NM, NOS, NS, NUMPAR, TLLOC, TLOC, TNCL, TNDI, TNFI, TNG, TNLG, TNLM, TNLS, TNM, TNOS, TNS

Name	Rules
Stylistic Issues	ESLINT_NMSAT
TS Best Practices	ESLINT_TSATH, ESLINT_TSBT, ESLINT_TSBTC, ESLINT_TSNEA, ESLINT_TSNENNA, ESLINT_TSNMN, ESLINT_TSNN, ESLINT_TSNNNANC, ESLINT_TSNR, ESLINT_TSPNK, ESLINT_TSTSR, ESLINT_TSUM
TS Possible Errors	ESLINT_TSNDCM, ESLINT_TSNEI, ESLINT_TSNEP, ESLINT_TSNES, ESLINT_TSNFIA, ESLINT_TSNFP, ESLINT_TSNIAC, ESLINT_TSNMP, ESLINT_TSNNNAOC, ESLINT_TSNUA, ESLINT_TSNUAS, ESLINT_TSNUCA, ESLINT_TSNUMA, ESLINT_TSNUR, ESLINT_TSPAC, ESLINT_TSRPO
TS Stylistic Issues	ESLINT_TSAOS, ESLINT_TSAT, ESLINT_TSNVR, ESLINT_TSUS
TS Variables	ESLINT_TSNMW, ESLINT_TSNTA, ESLINT_TSNUTA, ESLINT_TSNUTAS, ESLINT_TSNUTC, ESLINT_TSNUV
Variables	ESLINT_NDV, ESLINT_NUN

Definitions

Best Practices

These are rules designed to prevent you from making mistakes. They either prescribe a better way of doing something or help you avoid footguns.

Clone

Clone metrics: measure the amount of copy-paste programming in the source code.

Complexity

Complexity metrics: measure the complexity of source code elements (typically algorithms).

Coupling

Coupling metrics: measure the amount of interdependencies of source code elements.

Documentation

Documentation metrics: measure the amount of comments and documentation of source code elements in the system.

ECMAScript 6

These rules are only relevant to ES6 environments.

Inheritance

Inheritance metrics: measure the different aspects of the inheritance hierarchy of the system.

Possible Errors

These rules relate to possible syntax or logic errors in JavaScript code.

Size

Size metrics: measure the basic properties of the analyzed system in terms of different cardinalities (e.g. number of code lines, number of classes or methods).

Stylistic Issues

These rules are purely matters of style and are quite subjective.

TS Best Practices

These are rules designed to prevent you from making mistakes in TypeScript. They either prescribe a better way of doing something or help you avoid footguns.

TS Possible Errors

These rules relate to possible syntax or logic errors in TypeScript code.

TS Stylistic Issues

These rules are purely matters of style and are quite subjective within TypeScript.

TS Variables

These rules have to do with variable declarations within TypeScript.

Variables

These rules have to do with variable declarations.

The UserDefinedMetrics module

In addition to the built-in metrics, users can create their own metrics by using the UserDefinedMetrics (UDM) module. These new metrics can be established with a formula that can use any previous SourceMeter metric and can even depend on other custom user-defined metrics.

Defining a metric

- Metric definitions are surrounded by a `<tool name = "UDM" enabled = "true"></tool>` tag.
- Metrics are defined using a `<udm id = "NAME"></udm>` tag.
    - `NAME` follows standard identifier conventions (starts with a letter or an underscore, followed by any number of letters, numbers, or underscores).
    - `pi`, `epsilon`, and `inf` are occupied names (in a case-insensitive manner, so `Pi`, `pI`, `Epsilon`, `epSiloN`, etc. are occupied as well).
- Each metric definition can contain multiple configurations using a `<configuration name = "ConfigName"></configuration>` tag.
    - The `Default` configuration is mandatory, while the language-specific “overrides” are optional.
    - Each “override” inherits its starting setting values from the `Default` configuration.
    - The valid language-specific configuration values (at the moment) are `cpp`, `csharp`, `java`, `javascript`, `python`, and `rpg`.
- Each configuration can contain the following settings:
    - `description`: The description of the metric (default: “”).
    - `display-name`: An optional name, if something other than the ID should be used for display (default: “”).
    - `help-text`: Additional help text for the metric (default: “”).
    - `formula`: The formula used to calculate the value of the metric. It can be any valid expression supported by the ExprTk library and contain references to any previously computed metrics (e.g., LLOC, NOS, CBO, etc.).
    - `type`: The desired output type of the formula (valid values: “Integer”, “Float”; default: “Float”).
    - `calculated`: This field contains a list of syntactic element types (e.g., Class, Method, etc.) the metric is calculated for inside `<calculated-for>` tags. See the example below.
        - Note that every (both built-in and user-defined) metric has a unique set of elements it is computed for. Therefore, if a metric is dependent on another, its

`<calculated-for>` set should be a subset of the dependency's for it to be universally computable.

Example profile.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<analyzer-profile>
  <tool-options>
    <tool name = "UDM" enabled = "true">
      <udm id = "UDM_test_metric">
        <configuration name = "Default">
          <description>Blablabla</description>
          <display-name>UDM Test Metric</display-name>
          <help-text>A helpful comment</help-text>
          <formula>LLOC + 1 + sin(pi)^3</formula>
          <type>Integer</type>
          <calculated>
            <calculated-for>Class</calculated-for>
            <calculated-for>Method</calculated-for>
          </calculated>
        </configuration>
      </udm>
    </tool>
  </tool-options>
</analyzer-profile>
```

The UserDefinedPatterns module

As opposed to the UserDefinedMetrics module, where extra values could be calculated from already existing metrics, this module offers a way to tag source code elements (or a constellation of source code elements) that satisfy a certain set of arbitrary criteria. The use originally intended for this module is matching object-oriented design pattern and antipattern instances – but it can be used for discovering (or enforcing) any occurrence that can be expressed through the Language Independent Model the toolchain uses under the hood. Example pattern configurations for the most widespread design- and antipatterns can be found in `Common/Patterns`.

Usage

The module can be activated with a single extra switch (`-pattern=value`) to the toolchain. It can either be a single pattern, or a folder containing multiple patterns. Patterns can either be declarative (.yaml, .xml, or .json files), or procedural (.py python scripts), and in the case of folders, can be mixed arbitrarily. The declarative way is easier and possibly drier than writing code, so this is the primary entry point for users wishing to express a pattern It has its limitations, though, as the low-level formulas that do the actual matching can only use data that already exists in the model.

If a pattern needs extra calculations on the fly, it can use the procedural interface, and with it the full expressive capabilities of the python language. A binding interface is provided so that the python code can interact with the source code elements being visited so that it can calculate its own intermediary values and evaluate its own criteria. The binding also makes it possible for the pattern code to publish new values to the graph, as well as the main matches – thereby covering the expressive capabilities of the UserDefinedMetrics tool as well.

An example pattern

To get an idea how the two methods work, let's consider an example first. If we wanted to discover instances of the Singleton design pattern, we could phrase our requirements along the lines of:

- A class with only non-public constructors.
- A private static member attribute whose type is the class itself.
- A public static member function whose return type is the class itself.

When a class satisfying all these conditions is found, we expect an output like the following:

```
Pattern 'Singleton' has been found on 'SingletonExample'
Source : path/to/file.cpp [8,39,5,6]
Role 'SingletonClass' fits on 'SingletonExample' (nodeid '107'), path/to/file.cpp [8,39,5,6]
```

Additionally, this information would be tagged on the appropriate graph node, and be a part of the official output of the toolchain. Since the requirements of the Singleton pattern are relatively simple, we can start with the declarative method.

The declarative method

A usual pattern description consists of some metadata (pattern name, what kind of source code element we're looking for, what it is called within the pattern, etc.) and then a set of conditions we want to check. These conditions can be:

- Formulas, that are checked against the node in question.
- Context switches, where we can travel from the current node to another by the edges defined in the LIM model. For this, we can specify the edge to travel, the direction to travel in, the kinds of nodes we want to enumerate on the other end, what name we want to refer to these end nodes, and some nested conditions we want to match against them.
- Logical combinations, as in "and", "or", etc., where we can combine multiple formulas or context switches.

The best way to dive into the language is through an example. We could use either yaml, xml, or json, but yaml will be considered the default from now on. So, following the above guidelines, the Singleton description (with some additional, verbose annotation) would look something like this:

```
# we start by specifying the relevant metadata
# the names `name`, `kind`, `role`, etc.
# are all keywords so matches can automatically have their metadata added to them in the graph
name : Singleton
kind : ndkClass
role : SingletonClass
category: DesignPattern
priority: Info
description: >
  Application needs one, and only one, instance of an object.
  Additionally, lazy initialization and global access are necessary.
displayName: Singleton

# and then move on to the conditions we want to match
conditions:

  # 1, the "instance" condition, where we check that the class has a private static attribute whose type is the class itself
  # 2, this is implemented using a context switch to the class' member attributes, and then 3 nested conditions to check if
  #    the attribute is private, static, and matches the type of the class
  # 3, the context switch is an `any`, since we're already happy if at least one attribute satisfies our nested conditions
  #    -- but we could also use the other two context switches, `all` and `exists`
  # 4, everything we name in the `role` clause can then be used as a variable in any further condition
  # 5, note that `accessibility` and `isStatic` are built-in node characteristics from LIM, and ndkAttribute/ackPrivate
  #    are LIM constants
  # 6, to help bridge the divide between the logical and type hierarchies, we can refer to a node's type using the
  #    "node.type" shorthand
  - any :
      direction : forward
      edge : edkScope_HasMember
      kind : ndkAttribute
      role : attr1* # the trailing asterisk means that we don't want to include this attribute among the results
      conditions :
        - accessibility == ackPrivate
        - isStatic == true
        - type == SingletonClass.type
  # the "getInstance" condition, where we check that the class has a public static method that returns the class type
  # implementation is nearly identical to the above, only now we check ndkMethods instead of ndkAttributes
  # the `filter` clause pre-filters which nodes the subsequent conditions apply to -- in this case,
  # we are only looking for a normal method (so, not a constructor or destructor)
```

```
    - any :
        direction : forward
        edge : edkScope_HasMember
        type : ndkMethod
        role : m1*
        filters:
          - methodKind == mekNormal
        conditions :
          - isStatic == true
          - accessibility == ackPublic
          - returnType == SingletonClass.type
    # and finally the "constructor" condition, where we check that none of the class' contructors are public
    # here we use the `all` context switch because we need to make sure that all constructors match our non-public requirement
    # we also encounter the `or` combination condition to round out condition kinds (the others are `and`, and `not`)
    - all :
        direction : forward
        edge : edkScope_HasMember
        kind : ndkMethod
        role : m2*
        filters:
          - methodKind == mekConstructor
        conditions:
          - or :
            - accessibility == ackPrivate
            - accessibility == ackProtected
    # multiple conditions inside a `conditions` clause are implicitly joined by `and`, so all three of the above conditions
    # must be met for a class to be considered an instance of the Singleton pattern
```

With the mixture of formulas, context switches, and logical combinations, many patterns can be described.

Declarative reference

The basic analysis procedure usually leads to the following artifacts (among others) in increasing abstraction: Source code, language-specific model (ASG), language-independent model (LIM), result graph (graph). Since we are in language-independent territory here, the ASG does not concern us; but both the LIM and the graph are worth becoming familiar with, to help us fully utilize the capabilities of the pattern matching process.

LIM can be mostly thought of as the Object-oriented design level view of the whole system. It contains packages, classes, methods, attributes, and appropriate connections (edges) among these (for a full list of nodes and edges, please refer to the LIM introduction). It is a “lossy” conversion from the ASG, as the internals of the methods (so every statement, expression, local variable, etc.) is lost. But the overall structure remains, which is the important part for pattern detection.

The graph, on the other hand, is the result structure (the main output, so to speak) of the analysis process. It contains not only the corresponding nodes from the LIM, but also computed values like numeric metrics, source code position-based warnings, and other metadata to facilitate its many visual representations later. Pattern matches we discover also go onto the graph in the form of positioned warnings (possibly spanning across multiple nodes if the pattern has multiple roles). Additionally, we can use the graph as an “input” as well, and incorporate previously calculated values into the formulas of our pattern matching.

With this in mind, let’s see what the declarative method can offer us:

Metadata reference:

- `name` (**required**) : The name of the pattern, its value is scalar (string/text).
- `kind` (**required**) : The kind of the LIM node where the pattern matching begins, its value is scalar (string/text), refer to LIM’s `NodeKind`.
- `role` (**required**) : The name of the matched pattern on the given node, its value is scalar (string/text).
- `category` : Metadata for generating a rule for the pattern on the Graph, its value is scalar (string/text) [**DesignPattern**/**AntiPattern**/**CustomPatternName**]. By default its value is **DesignPattern**.
- `priority` : Metadata for generating a rule for the pattern on the Graph, its value is scalar (string/text) [**Info**, **Minor**, **Major**, **Critical**, **Blocker**]. By default its value is **Info**.
- `description` : Metadata for generating a rule for the pattern on the Graph. The description of the pattern.
- `displayName` : Metadata for generating a rule for the pattern on the Graph. The name of the pattern that will be visible in 3rd party programs.
- `filters` : Its value is a sequence, each entry is a condition. It filters the node, and only checks the conditions if the node satisfies the conditions described by the filter.
- `conditions` : Its value is a sequence, each entry is a condition. If this is not defined, the pattern matching will return true by default.

Context switches:

- `all`/`any` : Pattern matching returns true, if all/any of the conditions are matching, its values are maps (key/value pairs):
- `direction` : Sets the direction of the visiting, its value is scalar (string/text) [**forward**, **reverse**]. By default its value is **forward**.
- `from` : The starting node of the visiting, its value is scalar (string/text), can contain a roleName or/and class/type/parent hierarchy helpers (e.g., `from : myRoleName.parent.type`). By default its value is **this** (current node)
- `edge` (**required**) : The name of the edge, its value is scalar (string/text), refer to LIM’s `EdgeKind`.
- `kind` (**required**) : The kind of the LIM node where the pattern matching begins, its value is scalar (string/text), refer to LIM’s `NodeKind`.
- `role` (**required**) : The name of the matched pattern on the given node, its value is scalar (string/text).
- `filters` (not-required)
- `conditions` (not-required)

Logical combinations:

- `or` : Its value is a sequence, each entry is a condition (or a hierarchy switch). Returns true if any condition under it returns true.
- `and` : Its value is a sequence, each entry is a condition (or a hierarchy switch). Returns true if every condition under it returns true.
- `xor` : Its value is a sequence, each entry is a condition (or a hierarchy switch). Returns true if the number of true conditions under it is an odd number.
- `not` : Negate a single condition
- `nor` : Like “or” but negated
- `nand` : Like “and” but negated
- `exists` : Its values are maps (key/value pairs), checks whether the given edge exists on the given node or not:
- `direction` : Sets the direction of the visiting, its value is scalar (string/text) [**forward**, **reverse**]. By default its value is **forward**.
- `from` : The starting node of the visiting, its value is scalar (string/text), can contain a roleName or/and class/type/parent hierarchy helpers (eg., `from : myRoleName.parent.type`). By default its value is **this** (current node)
- `edge` (**required**) : The name of the edge, its value is scalar (string/text), refer to LIM’s `EdgeKind`.

List of exact formula capabilities

Any value that is available on the Graph. Refer to the whole documentation for what various tools output to the graph, and how. We recommend looking into the Source Code Metrics first. Referable LIM values:

Reference name	Return type	FileSystem	File	Package	Component	Scope	Class	Attribute	Method	Parameter	Type	Every Node	Description
id	int											X	Returns the ID of the node as a number
parameterSize	int								X				Returns the number of parameter the method has
isStatic	bool			X		X	X	X	X				Returns true/false if the LIM node is static
isVirtual	bool								X				Returns true/false if the LIM node is virtual
isAbstract	bool						X		X				Returns true/false if the LIM node is abstract

Reference name	Return type	FileSystem	File	Package	Component	Scope	Class	Attribute	Method	Parameter	Type	Every Node	Description
isAnonymous	bool			X		X	X		X				Returns true/false if the LIM node is anonymous
accessibility	string			X		X	X	X	X				Returns the accessibility `AccessibilityKind` as a text
classKind	string						X						Returns the kind of a class `ClassKind` as a text
methodKind	string								X				Returns the kind of a method `MethodKind` as a text
name	string	X	X	X	X	X	X	X	X	X			Returns the name of the node as a text
kind	string											X	Returns the kind of a node `NodeKind` as a text
paramKind	string									X			Returns the kind of a parameter `ParameterKind` as a text
simpleTypeKind	string										X		Returns the type of a simpleKind `SimpleTypeKind` as a text
pointerKind	string										X		Returns the kind of a `PointerKind` as a text
returnType	LIM::Node								X				A logical::Type node of the method returns
type	LIM::Node						X	X	X	X			A logical::Type node
class	LIM::Node										X		A logical::Class node
parent	LIM::Node						X	X	X	X			A logical::… node of the parent

With these values, arbitrary expressions can be created using basic arithmetic operators (`+`, `-`, `*`, `/`) and boolean comparisons (`==`, `!=`, `~=`, `>`, `>=`, `<`, `<=`).

Commonly used context switches

- `edkScope_HasMember` : from classes to member methods/attributes (and back).
- `edkClass_IsSubclass` : from classes to their descendants (and back).
- `edkMethod_HasParameter` : from methods to their parameters (and back).
- …for a complete list, refer to the LIM introduction post.

A more complex pattern

There can be instances, however, where the declarative method is just not expressive enough. For example, when we need more complex combinations than "any" or "all". Or when a temporary value needs to be computed before a condition could be evaluated.

An example taking advantage of this increased expressive power is the Feature Envy antipattern, where our match requirement can be summarized like the following:

- First, we need to calculate the occurrences of local and foreign attribute accesses.
- Then the condition is to check whether there are "enough" attribute accesses, and if so, whether the percentage of foreign accesses is above a certain threshold.

The procedural method

For these cases, there's also a python binding for the whole pattern matching process, where a custom python script can access the source code elements through a standardized node interface. This interface provides ways to read the same values, to switch contexts through the same edges, or to manually place pattern matches on the graph. But on top of this, it also provides free reign to calculate arbitrary values, iterate subsets, perform conditional execution, etc. A script for Feature Envy would look something like this:

```
# the same metadata, only now in the form of "global" variables
name = "featureEnvy"
role = "envyMethod"
kind = "ndkMethod"
priority = "Minor"
category = "AntiPattern"
displayName = "Feature Envy"
description = "A method accesses the data of another object more than its own data."

# the `visit` method is the entry point for the pattern matching, which is called for each node of the appropriate type
# nodes here are of a special NodeBinding type that allows python access to everything we've seen from the declarative section
# only now, we can use that data to perform arbitrary calculations
def visit(node):
 parent = node.getParent()
 localAccess = 0
 foreignAccess = 0
 # context switches are now done using the `traverse` method
 for methodAccAttr in node.traverse(edge = "edkMethod_AccessesAttribute", reverse = False):
  for attrAccAttr in methodAccAttr.traverse(edge = "edkAttributeAccess_Attribute", reverse = False):
   for methodUsesThisClass in attrAccAttr.traverse(edge = "edkScope_HasMember", nodeKind = "ndkClass", reverse = True):
    # but once the destination is reached, we can increment counters
    if parent == methodUsesThisClass:
     localAccess += 1
    else:
     foreignAccess += 1

  # if the number of attribute accesses surpasses a certain limit (5, in this case), and at least 80% of those were
  # directed towards a foreign attribute, we officially call the class "envious"
  if (localAccess + foreignAccess >= 5) and (foreignAccess / (foreignAccess + localAccess) >= 0.8) :
    node.setWarning(
     patternName = name,
     roleName = role,
     category = category,
     description = description,
     displayName = displayName,
     priority = priority
    )
```

Of course, anything that could be done with the declarative method can also be done using the procedural method. As an illustration, the Singleton pattern from above could be rewritten like so:

```
name = "Singleton"
role = "Singleton"
kind = "ndkClass"
category = "DesignPattern"
priority = "Info"
description = """Application needs one, and only one, instance of an object. Additionally,
                  lazy initialization and global access are necessary."""
displayName = "Singleton"

def visit(node):

 hasInstance = False
 for attr1 in node.traverse("edkScope_HasMember", nodeKind = "ndkAttribute"):
  if (
    attr1.getValue("accessibility") == "ackPrivate" and
    attr1.getValue("isStatic") == True and
    attr1.type_equals(node)
   ):
   hasInstance = True
   break

 getInstance = False
 for m1 in node.traverse("edkScope_HasMember", nodeKind = "ndkMethod"):
  if (
    m1.getValue("methodKind") == "mekNormal" and
    m1.getValue("accessibility") == "ackPublic" and
    m1.getValue("isStatic") == True and
    m1.type_equals(node)
   ):
   getInstance = True
   break

 constr = True
 for m2 in node.traverse("edkScope_HasMember", nodeKind = "ndkMethod"):
  if m2.getValue("methodKind") != "mekConstructor": continue
  if m2.getValue("accessibility") == "ackPublic":
   constr = False
   break

 if hasInstance and getInstance and constr:
  node.setWarning(name, role, displayName, category, priority, description)
```

Procedural reference

There are two distinguished top-level names for pattern-matching python scripts:

1. `kind`: which is the same as the top-level `kind` attribute was for the declarative method, namely it pre-filters the node kinds being visited when checking for this pattern.
2. `visit`: which is the entry point for the evaluation of each (appropriate) source code node in the input program. Its only parameter is the node currently being visited, in the form of a `NodeBinding` object, described below. Optionally, any value returned from this visit function (while otherwise ignored) will be logged to the standard output for debugging purposes.

The `NodeBinding` objects can be interacted with using the following methods:

- `getValue(name)`: the main way to read data from the current node. First, it looks for distinguished keywords (please refer to the declarative method for a complete list), and if it can't find the reference there, it looks for an arbitrary Graph node attribute with the same name. This method can therefore be used to retrieve both LIM and Graph values.
- `setMetric(name, value)`: the pair of `getValue`, only it writes data instead of retrieving it. It is important to note, though, that it cannot alter any LIM-related data, and can only set attributes in the Graph.
- `getType()` / `getClass()` / `getParent()`: the same bridge between the logical and type hierarchies that was provided as `.type`, `.class` and `.parent` in the declarative method. Returns another `NodeBinding` referring to the starting node's type / class / parent, respectively.
- `traverse(edge, kind, reverse=False)`: performs the previously mentioned context switches between nodes by traversing specific edges. `kind` can be specified if we only want to traverse those kinds of nodes on the other end of the given edges. `reverse` reverses the direction of the edge traversal, listing the nodes that point to this node by the given edge, not the nodes this node points to.
- `typeEquals(other)` / `typeSimilar(other)`: performs type-related checks. `typeEquals` is true only if the current node's type and the type of the `other` parameter are the same, while `typeSimilar` is also true if they have some heuristic-based connection. For example, if one is a custom class, and the other is a list containing custom class instances.
- equality / inequality checks are overridden for `NodeBindings`, so identity checks can be performed using the usual `==` and `!=` operators.
- `setWarning(name, role, displayName, category, priority, description, children)`: sets match results on the Graph. `name`, `role`, `displayName`, `category`, `priority`, and `description` are just the usual metadata accompanying the match, while `children` is an optional python `dict` containing the sub-roles of the pattern.

For an example of sub-roles, consider the Composite pattern, where we might want to indicate the Component and Leaf participants of the pattern as well. In this case, we might call `setWarning` something like this:

```
node.setWarning('Composite', 'Composite', 'Composite', 'DesignPattern', 'Minor', 'lorem ipsum', {
  'Component': component_node,
  'Leaf': [leaf_node_1, leaf_node_2]
})
```

According to the above, sub-roles can be individual nodes or node lists, which will refer back to the original pattern match.

Predefined Design patterns

The SourceMeter tool comes with a catalogue of predefined design patterns, listed below.

Abstract Factory

A pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Interpreted as: a stateless abstract class with at least two abstract methods, both of which return an abstract type. Furthermore, we check the existence of at least two derived classes that implement those two abstract methods of the base class, and instantiate concrete objects *derived* from original return types. The former condition (two or more abstract methods) refers to the "families of related objects" part, while the latter (instantiation of derived objects instead of the original return values) is the characteristic of factories.

Adapter

A pattern acting as a wrapper between two objects, catching calls for one object and transforming them to a format and interface recognizable by the second object.

Interpreted as: a class that meets all three of the following conditions:

- it implements an interface (showing that it is trying to adapt *to* something),
- it contains a class-type attribute that does **not** implement the same interface (showing that it is trying to adapt *from* something), and
- it has a constructor with the adaptee's type (showing that it is created by wrapping something).

Bridge

A pattern that decouples an abstraction from its implementation so that the two can vary independently. It represents the connection of two different inheritance hierarchies through composition.

Interpreted as: the head of the first class hierarchy (so at least two child classes for interfaces, or one child for instantiable classes), where:

- it has a private attribute referencing the head of the second, different hierarchy (again, with at least two child classes for interfaces, or one child for instantiable classes), and
- a contructor and at least one "normal" method that uses this attribute.

Builder

A pattern that separates the construction of a complex object from its representation so that the same construction process can create different representations.

Interpreted as: an abstract class that meets all of the following conditions:

- doesn't have any attributes,

- has at least two abstract methods,
- has at least one child class containing a “build” method (that returns a private attribute) and “reset” method (whose name is checked semantically), and
- is used in a “Director” class, where
    - it’s a private attribute,
    - used by at least two different methods, and
    - the different methods access it a different amount of times.

Chain of Responsibility

A pattern that makes it possible to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. It “chains” the receiving objects and passes the request along the chain until an object handles it.

Interpreted as: an abstact class that satisfies all of the following conditions:

- contains a private attribute with its own type (referring to the next handler in the chain),
- has a setter for this attribute,
- can recursively call itself on this private attribute, and
- it has at least two child classes that override the specific handling logic.

Composite

A pattern that composes objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Interpreted as: an abstract class or interface (called Component) that has multiple child classes, at least one of which contains a private attribute of the Component’s type.

Object Pool

A pattern that can offer a significant performance boost in situations where 1) the cost of initializing a class instance is high, 2) the rate of instantiation of a class is high, and 3) the number of instantiations in use at any one time is low. It achieves this by maintaining a limited pool of reusable instances.

Interpreted as: a class that satisfies all of the following conditions:

- has at least one attribute (referencing the object pool itself),
- all of its attributes are private (so that we’re forced to use the sharing interface),
- has at least five methods (related to the getting/releasing/capacity settings of the pool),
- and extends the Singleton requirements (private constructor, public static instance getter).

Prototype

A pattern that specifies the kinds of objects to create by using a prototypical instance, and creates new objects by copying this prototype.

Interpreted as: a class that has a virtual cloning/copying method (according to its naming convention) that returns its own type, and at least one child class that overrides this method with a covariant return type.

Singleton

A pattern that ensures that a class has only one instance, and provides a global point of access to it, thereby achieving lazy (or “just-in-time”) initialization.

Interpreted as: a class that safisfies all three of the following conditions:

- it only has private constructors, thereby making it impossible for users to instantiate normally,
- it has a private, static field of its own type (supposedly containing the single instance everyone shares), and
- it has a public, static getter method that accesses the instance attribute and returns its type.

Predefined Anti patterns

The SourceMeter tool comes with a catalogue of predefined antipatterns, listed below.

Feature Envy

Happens when a method accesses the data of another object more than its own data.

Interpreted as: a class that accesses at least 5 attributes, and the ratio of foreign attributes among these (as opposed to local ones) exceeds 80%.

Long Method

A method that contains too many lines of code or is too complex (and therefore should be broken up into more manageable pieces).

Interpreted as: a function or method where either of the following three conditions are met:

- logical lines of code exceed 80,
- number of statements exceeds 80, or
- the McCabe complexity exceeds 10.

Long Parameter List

A method that has too many parameters, usually signaling that multiple algorithms have been merged, or that subsets of the parameter list should be encapsulated into their own class.

Interpreted as: a method that has more than 6 parameters.

Refused Bequest

Happens when a subclass uses only some of the methods and properties it inherited from its parents, which usually signals that inheritance was the incorrect choice of code reuse (and should be replaced with delegation).

Interpreted as: A class that inherits protected attributes or methods that it doesn’t ever access.

Shotgun Surgery

Happens when making any functionally coherent modification requires that developers make many small changes to many different classes. Usually signals that the class(es) implementing the feature are too closely coupled to other parts of the system.

Interpreted as: A method that has more than 10 incoming invocation.

The Blob

Also known as God Class or Large Class Code, the blob is a class that monopolizes the processing, while other classes primarily encapsulate data only. It is a thin layer of object orientation to hide a design philosophy that is essentially still procedural.

Interpreted as: a class where:

- the combined number of methods and attributes exceeds 60, or
- the logical lines of source code exceed 1000.

Footnotes