
OCLint Documentation

Release 20.11

Longyi Qi, Ryuichi Laboratories

Nov 18, 2020

Contents

1	Overview	1
1.1	Features	1
1.2	Current Status	1
1.3	Get Started	2
1.4	Get Involved	2
2	Introduction	3
2.1	Download	3
2.2	Building OCLint	3
2.3	Installation	4
2.4	Homebrew	6
2.5	Tutorial	6
2.6	Directions to Safe Use	9
3	Manual	11
3.1	oclint Manual	11
3.2	oclint-json-compilation-database Manual	15
3.3	oclint-xcodebuild Manual	16
4	User Guide	19
4.1	Using OCLint with CMake	19
4.2	Using OCLint with Bear	20
4.3	Using OCLint with xcodebuild	21
4.4	Using OCLint with xctool	24
4.5	Using OCLint with xcpretty	25
4.6	Using OCLint in Xcode	25
4.7	Using OCLint with Travis CI	29
4.8	Using OCLint with Jenkins CI	30
5	How-To	37
5.1	Customize Rule Thresholds	37
5.2	Select OCLint Rules for Inspection	38
5.3	Pick Up the Right Reporter	39
5.4	Suppress OCLint Warnings	41
5.5	Preserve Configurations	43
5.6	Set Up OCLint in MinGW Environment	44

6	Rule Index	47
6.1	Basic	47
6.2	Cocoa	54
6.3	Convention	56
6.4	Design	62
6.5	Empty	63
6.6	Migration	66
6.7	Naming	68
6.8	Redundant	69
6.9	Size	72
6.10	Unused	77
7	OCLint Internals	79
7.1	Core Module	79
7.2	Metrics Module	80
7.3	Rules Module	80
7.4	Reporters Module	81
8	Development	83
8.1	System Requirements	83
8.2	Checking Out Code	84
8.3	Compiling and Testing	87
8.4	Writing Custom Rules	90
8.5	Writing Custom Reporters	92
8.6	Scaffolding	93
8.7	Eating Your Own Dog Food	95
8.8	Open Projects	96
8.9	Related Clang Knowledge Base	98
8.10	Coding Standards	99
9	Modified BSD License	101
	Bibliography	103

OCLint is a [static code analysis](#) tool for improving quality and reducing defects by inspecting C, C++ and Objective-C code and looking for potential problems like possible bugs, unused code, complicated code, redundant code, code smells, bad practices, and so on.

1.1 Features

Static code analysis is a critical technique to detect defects that aren't visible to compilers. OCLint automates this inspection process with advanced features:

- Relying on the [abstract syntax tree](#) of the source code for better accuracy and efficiency; False positives are mostly reduced to avoid useful results sinking in them.
- Dynamically loading rules into system, even in the runtime.
- Flexible and extensible configurations ensure users in customizing the behaviors of the tool.
- Command line invocation and configuration persistence facilitate continuous inspection of the code while being developed, so that technical debts can be fixed early on to reduce the maintenance cost.

1.2 Current Status

OCLint is far from finished but is being continuously improved in many aspects, e.g. accuracy, performance and usability.

OCLint started as a [research project](#) at the [University of Houston](#). Since then, OCLint has been rewritten and grown to be a 100% open source project. OCLint is designed to serve both academia and industry. Our goal is to spread the importance of code quality, and to make OCLint a must-have tool for developers who program in C, C++ and Objective-C languages.

OCLint is compatible with macOS, major BSD and Linux variants. Porting to Windows is experimental with community effort.

1.3 Get Started

There is a good chance that pre-compiled binaries are ready to [download](#).

It's also quite easy to start with [getting the source code](#) and [building it](#) locally in a specific environment.

1.4 Get Involved

Please also consider [getting involved](#) in the OCLint community. All kinds of contributions, like giving feedback, starting a discussion, filing issues, requesting new features, best of all, submitting a patch or two, are sincerely appreciated.

2.1 Download

All release downloads can be found at [OCLint GitHub Releases page](#).

2.1.1 Choosing Pre-Compiled Binaries

Pre-compiled binaries are considered to ease you get started. OCLint binaries depend on very little and fundamental system standard libraries. Therefore, each pre-compiled bundle archive is expected to work on the platform it specifies.

In case the compiled binary doesn't work on this particular environment, consider [building OCLint locally](#).

See [installation](#) document for next step when download process is finished.

2.1.2 Choosing Source Code

All source code releases contain every module that is required for compilation.

With development codebase, all dependencies are caught up to the latest version; on the contrary, with released codebase, everything is finalized and optimized for that particular version.

Read [how to build OCLint](#) when the source code is on hand.

2.2 Building OCLint

This page presents building OCLint in release mode.

2.2.1 System Requirements

1. See [LLVM System Requirements](#)
2. [Python](#)
3. [Git](#)
4. [Apache Subversion](#)
5. [CMake](#)
6. [OpenSSL](#) (only for analytics-enabled build)

2.2.2 Building OCLint

The whole build process can be done with the make script inside `oclint-scripts` folder.

1. `cd oclint-scripts`
2. `./make`
3. `cd ..`

2.2.3 Verifying the Build

By calling the binary

```
./build/oclint-<major>.<minor>.dev.<git-hash>/bin/oclint
```

The following error message is expected:

```
oclint: Not enough positional command line arguments specified!
Must specify at least 1 positional arguments: See: ./oclint -help
```

It's highly recommended to [add the bin directory into system PATH](#).

See also:

If you are looking for compiling and testing OCLint in debug mode for development and testing purposes, please move onto [development](#) section.

2.3 Installation

Both pre-compiled binary distribution and local build bundle should end up with an OCLint release with which file tree similar to this:

```
oclint-release
|-bin
|-lib
|---clang
|-----<llvm/clang version>
|-----include
|-----lib
|---oclint
|-----rules
```

(continues on next page)

(continued from previous page)

```
|-----reporters
|-include
|---c++
|-----v1
```

Even without installation, `oclint` is able to be invoked directly from `bin` directory now.

In order to ease the invocation, it's recommended to add OCLint's `bin` folder to system `PATH`, the environment variable that tells system which directories to search for executable files.

2.3.1 Option 1: Directly Adding to PATH

Following code snippet is an example for the `.bashrc` or `.bash_profile` file that is sourced when terminal launches.

```
OCLINT_HOME=/path/to/oclint-release
export PATH=$OCLINT_HOME/bin:$PATH
```

2.3.2 Option 2: Homebrew Tap

macOS users can install [our homebrew tap](#)

2.3.3 Option 3: Copying OCLint to System PATH

A few directories are supposed to be in the system `PATH` already, to mention a few, `/usr/local/bin`, `/usr/bin`, `/bin`, etc. Therefore, it's also possible to copy the OCLint binaries into one of these folders, and move the dependencies over. As an example, presumes `/usr/local/bin` is in the `PATH` (may require root permission).

1. `cp bin/oclint* /usr/local/bin/`
2. `cp -rp lib/* /usr/local/lib/`
3. `cp -rp include/* /usr/local/include/`

Dependency libraries are required to be put into appropriate directory, because `oclint` executable searches `$(/path/to/bin/oclint)/../lib/clang`, `$(/path/to/bin/oclint)/../include/c++`, `$(/path/to/bin/oclint)/../lib/oclint/rules` and `$(/path/to/bin/oclint)/../lib/oclint/reporters` for builtin headers and dynamic libraries by default.

Warning: Since you are making changes to the system `PATH` directly, please be advised that it may break the dependencies for other tools or system libraries. If you don't have confidence, please use other installation options.

2.3.4 Verifying Installation

Open a new terminal prompt, and execute `oclint` directly from there and expect message similar to below:

```
$ oclint
oclint: Not enough positional command line arguments specified!
Must specify at least 1 positional arguments: See: oclint -help
```

That's it – if OCLint is pretty new to you, [tutorial](#) would lead you by applying the tool to a sample code, and explaining a few concepts along the way.

2.4 Homebrew

macOS users can install and update OCLint from our [homebrew tap](#).

2.4.1 Installing OCLint

```
$ brew tap oclint/formulae
$ brew install oclint
```

2.4.2 Updating OCLint

```
$ brew update
$ brew upgrade oclint
```

2.5 Tutorial

This tutorial walks you through the inspection of a piece of small but smelly C++ source code. By the end of this tutorial, you should be able to

- apply OCLint on a single file
- configure OCLint with very basic compiler flags
- understand the output

Throughout this tutorial, we will also lead you to the detail pages if you are interested in certain steps and are willing to know more.

2.5.1 Something Smells Here

Create a `sample.cpp` file with the content below:

```
int main() {
    int i = 0, j = 1;
    if (j) {
        if (i) {
            return 1;
            j = 0;
        }
    }
    return 0;
}
```

2.5.2 Building Sample Code (Optional)

It is actually not necessary to build the code prior to run OCLint against it. However, since finding the correct arguments becomes one of the most frequently asked questions, this step is trying to help convert the compiler flags to the ones that OCLint requires.

Note: This step, however, doesn't intent to show how to find the correct compiler flags, thus, some level of knowledge about compiler flags is a prerequisite.

```
$ CC -c sample.cpp // step 1: compiling generates sample.o
$ CC -o sample sample.o // step 2: linking generates sample executable file

// Change CC to your favorite compiler that is GCC-compatible, e.g. g++ and clang++

$ ./sample // execute the binary
$ echo $? // output of 0 probably means the code has been successfully built
```

We just took two sequential steps to generate the binary, step 1 compiles the code, and step 2 links. We are only interested in step 1 because that's all compiler flags we need to give to OCLint. Here in this case, the compiler flag is `-c`, and inspected source file is `sample.cpp`.

If you cannot pass through this step, don't give up, there are some tools try to help, like [CMake](#) and [Bear](#) (for Make). In addition, we also provide two helper programs [oclint-json-compilation-database](#) and [oclint-xcodebuild](#) (for Xcode users) could help find the arguments for OCLint.

2.5.3 Checking Single File

OCLint checks a single file using the following format:

```
oclint [options] <source> -- [compiler flags]
```

So, the command that applies to the sample source is

```
$ oclint sample.cpp -- -c
```

To change OCLint behavior, change the `[options]` before the source; to alter the compiler behavior, change the `[compiler flags]` after the `--` separator. A complicated example might look like this:

```
$ oclint -report-type html -o report.html sample.cpp -- -D__STDC_CONSTANT_MACROS -D__
↳STDC_LIMIT_MACROS -I/usr/include -I/usr/local/include -c
```

For detail about OCLint options, see [oclint manual](#).

2.5.4 For Projects with Multiple Files

The approach described in the previous section works perfectly for a single file or a few files. The inspection process is fast, and making changes to arguments is easy.

While working on a project with a group of source files, inspecting the entire project at once and having a single report is preferred.

When all sources share the same compiler flags, we can do

```
oclint [options] <source0> [... <sourceN>] -- [compiler flags]
```

However, each source file may have different compiler flags. In this case, by reading from the **compilation database**, OCLint can recognize the list of source files for analysis, along with the compiler flags used for each time during the compilation phase. It can be considered as a condensed Makefile. So, in this case

```
oclint -p <build-path> [other options] <source0> [... <sourceN>]
```

A more handy helper program that comes with OCLint is `oclint-json-compilation-database`. If you use OCLint to analyze projects, for most of the time, you will deal with `oclint-json-compilation-database` instead, and indirectly talk to `oclint`.

For people who work on a Mac with Xcode as IDE, you may find [Using OCLint with xcodebuild](#) and *Using OCLint in Xcode* <../guide/xcode.html> documents are helpful.

We also provide guidances for people who use `CMake` and `make` as their build system.

2.5.5 Understanding Report

By applying OCLint against the above sample, with the default text reporter, the output is similar to this:

```
Processing: /path/to/sample.cpp.
OCLint Report

Summary: TotalFiles=1 FilesWithViolations=1 P1=0 P2=1 P3=1

/path/to/sample.cpp:4:9: collapsible if statements P3
/path/to/sample.cpp:9:17: dead code P2

[OCLint (http://oclint.org) v0.8]
```

Basically, the following information can be found in the report:

- Summary
 - total files
 - files with violations
 - number of priority 1 violations
 - number of priority 2 violations
 - number of priority 3 violations
- A list of violations
 - path to the source file
 - line number
 - column number
 - violated rule
 - priority
 - message (if any)
- Compiler diagnostics
 - compiler errors if any

- compiler warning if any
 - clang static analyzer results when it is enabled
- OCLint information
 - website
 - release version

Read more about [picking up the right reporter](#).

Next, more detail information can be found with comprehensive [manuals](#) and [user guides](#). In addition, a few [how-to documents](#) can help speed things up a little bit in several aspects.

2.6 Directions to Safe Use

We are drafting this document by following the suggestions from [Andy Hunt's proposal](#).

There are many many cool features that we can provide. At the same time, there are restrictions that we currently don't have a solution but only workarounds. We, as contributors, would like to be honest about the flaws in our tool. We hope these directions to safe use could be beneficial for both OCLint users and contributors -

- Users can save time by avoiding falling into the same issues that has already trapped other users for a long time but currently no solution due to the constraints by the tool itself
- Contributors, on the other hand, can save efforts by avoiding spending time digging into different aspects but turn out to be the same issue. By saving the time, contributors can use the passion on experimenting new ideas, improving existing features, and fixing the true problems

Alright, let's go with a few -

- [No white spaces in file name and path](#)
 - The existence of white spaces can cause problem when use `oclint-json-compilation-database` and `oclint-xcodebuild`

We hope you'll enjoy OCLint, and can get benefits from using it. Look forward to your suggestions. Thank you.

3.1 oclint Manual

When we invoke OCLint, it normally does rule loading, compilation, analysis, and report generation. The options allow us to change the behavior of each step to certain ways that meet our requirements.

See all supported options in OCLint 20.11 by typing `oclint -help`:

```

USAGE: oclint [subcommand] [options] <source0> [... <sourceN>]

OPTIONS:

Generic Options:

-help                      - Display available options (-help-hidden for more)
-help-list                 - Display list of available options (-help-list-hidden_
↳ for more)
-version                  - Display the version of this program

OCLint options:

-R=<directory>            - Add directory to rule loading path
-allow-duplicated-violations - Allow duplicated violations in the OCLint report
-disable-rule=<rule name>  - Disable rules
-enable-clang-static-analyzer - Enable Clang Static Analyzer, and integrate results_
↳ into OCLint report
-enable-global-analysis    - Compile every source, and analyze across global_
↳ contexts (depends on number of source files, could results in high memory load)
-extra-arg=<string>        - Additional argument to append to the compiler command_
↳ line
-extra-arg-before=<string> - Additional argument to prepend to the compiler_
↳ command line
-list-enabled-rules        - List enabled rules
-max-priority-1=<threshold> - The max allowed number of priority 1 violations

```

(continues on next page)

(continued from previous page)

```

-max-priority-2=<threshold> - The max allowed number of priority 2 violations
-max-priority-3=<threshold> - The max allowed number of priority 3 violations
-no-analytics               - Disable the anonymous analytics
-o=<path>                   - Write output to <path>
-p=<string>                 - Build path
-rc=<parameter>=<value>    - Override the default behavior of rules
-report-type=<name>        - Change output report type
-rule=<rule name>          - Explicitly pick rules

```

-p <build-path> **is** used to read a **compile** command database.

For example, it can be a CMake build directory **in** which a file named `compile_commands.json` exists (use `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` CMake option to get this output). When no build path **is** specified, a search **for** `compile_commands.json` will be attempted through **all** parent paths of the first **input** file. See: <http://clang.llvm.org/docs/HowToSetupToolingForLLVM.html> **for** an example of setting up Clang Tooling on a source tree.

<source0> ... specify the paths of source files. These paths are looked up **in** the **compile** command database. If the path of a file **is** absolute, it needs to point into CMake's source tree. If the path is relative, the current working directory needs to be **in** the CMake source tree **and** the file must be **in** a subdirectory of the current working directory. `./` prefixes **in** the relative files will be automatically removed, but the rest of a relative path must be a suffix of a path **in** the **compile** command database.

For more information, please visit <http://oclint.org>

3.1.1 Rule Loading Options

-R <directory> Rule loading path can be changed by using `-R` option. Multiple rule loading paths can be specified to load rules from more than one directories. By default, OCLint searches `$(/path/to/bin/oclint)/. /lib/oclint/rules` for the dynamic libraries that contain rules.

-disable-rule <rule name> This option gives the capability to deactivate rules by their names even though they are loaded into the system. This option can be chained multiple times to further narrow down.

-rc <parameter>=<value> Certain rules have threshold to decide whether to emit violations. These thresholds can be changed by `-rc` option with a key-value pair.

More detail on changing the behavior in rules loading process during runtime can be found in [rule selection](#) and [threshold customization](#) pages.

3.1.2 Compilation Options

For the sources that are being inspected, OCLint needs to know the compiler options for each of them. These options are the same arguments for actual compilers, like `gcc` or `clang`. There are two alternatives for passing the options to OCLint, namely specifying the compiler options directly to OCLint and using compilation database.

Giving Compiler Options

Compiler options can be given directly to OCLint. It's quite straight forward, after all OCLint options and sources, append `--` separator followed by all compiler options:

```
oclint [oclint options] <source0> [... <sourceN>] -- [compiler options]
```

For example, if we are compiling a file by the following `clang` command:

```
clang -x objective-c -arch armv7 -std=gnu99 -fobjc-arc -O0 -isysroot /Developer/SDKs/iPhoneOS6.0.sdk -g -I./Pods/Headers -c RPAActivityIndicatorManager.m
```

(Wow, it's longer than expectation.)

Then when we analyze this code, our OCLint command will be:

```
oclint [oclint options] RPAActivityIndicatorManager.m -- -x objective-c -arch armv7 -std=gnu99 -fobjc-arc -O0 -isysroot /Developer/SDKs/iPhoneOS6.0.sdk -g -I./Pods/Headers -c
```

Compilation Database

-p <build-path> Choose the build directory in which a file named `compile_commands.json` exists. When no build path is specified, a search for `compile_commands.json` will be attempted through all parent paths of the first input file.

If no compiler options are given explicitly, OCLint requires this compilation database to understand specific build options for each source file. Currently it supports `compile_commands.json` file. See [oclint-json-compilation-database](#) for detail.

3.1.3 Sources Options

We specify the path to all the source files we want to inspect. Multiple files can be analyzed with one invocation.

3.1.4 Report Options

-o <path> Instead of piping output to console, `-o` will redirect the report to the `<path>` you specified.

-report-type <name> Change output report type, by default, plain text report is used

See [picking up the right reporter](#) for detail.

3.1.5 Exit Status Options

-max-priority-1 <threshold> The max allowed number of priority 1 violations

-max-priority-2 <threshold> The max allowed number of priority 2 violations

-max-priority-3 <threshold> The max allowed number of priority 3 violations

This option helps continuous integration and other build systems. OCLint returns with one of the five exit codes below

- 0 - SUCCESS
- 1 - RULE_NOT_FOUND

- **2** - REPORTER_NOT_FOUND
- **3** - ERROR_WHILE_PROCESSING
- **4** - ERROR_WHILE_REPORTING
- **5** - VIOLATIONS_EXCEED_THRESHOLD
- **6** - COMPILATION_ERRORS

OCLint always return code zero for success execution with the number of violations under an acceptable range. Exit code other than zero means there are something wrong.

For example, when the compilation process fails, an exit code of 3 will be returned. It may means the compiler options have not been set correctly. When the source code has errors, the exit code will be 6.

When the number of violations in any of the priorities is larger than the maximum tolerance, OCLint returns with an exit status code of 5. By default, less than 20 priority 3 violations are allowed, 10 violations is maximum for priority 2, and no priority 1 violation can be tolerated. Too many violations result in bad code quality, if that happens, OCLint intends to fail the build system.

3.1.6 Global Analysis Options

-enable-global-analysis enable OCLint global analysis

With global analysis enabled, entire context of all given source code files is hold in the memory, and is available to the current analyzing file. This enables metrics calculation and other analyses that require cross-reference of the other files in the same project, which improves the accuracy of the analysis. However, global analysis results in high memory load, and may end up with long analysis duration, so it's designed for powerful machines and is enabled only by users' intents.

3.1.7 Clang Static Analyzer Options

-enable-clang-static-analyzer enable Clang Static Analyzer

When Clang Static Analyzer is enabled, OCLint will invoke it under the hook along with the process, collect its results, and emit them with the reporter. Notice that, by invoking Clang Static Analyzer, it will significantly increase the total analysis time.

3.1.8 Debug Options

-debug invoke OCLint in debug mode.

If OCLint is built in the debug model, `-debug` outputs deeper message from OCLint invocation. It prints messages that can help understand the overall progress of OCLint analysis.

Note: Please aware that this is only available when OCLint is built with debug flag on.

3.1.9 Other Options

-no-analytics Disable the anonymous analytics.

-version Show version information about OCLint, LLVM and some environment variables.

3.2 oclint-json-compilation-database Manual

`oclint-json-compilation-database` is a helper program that eases running OCLint against a set of source code files. Instead of specifying compiler flags for each file when `oclint` is invoked, these compiler flags could be collected and persistently saved to `compile_commands.json` file in JSON Compilation Database format.

See also:

Please read [JSON Compilation Database Format Specification](#) for detail.

`oclint-json-compilation-database` can fetch necessary information from `compile_commands.json` and kick off `oclint` with these compiler flags under the hook for code analysis. See the usage by typing `oclint-json-compilation-database -help`:

```
usage: oclint-json-compilation-database [-h] [-v] [-debug] [-i INCLUDES]
                                         [-e EXCLUDES]
                                         [oclint_args [oclint_args ...]]

OCLint for JSON Compilation Database (compile_commands.json)

positional arguments:
  oclint_args            arguments that are passed to OCLint invocation

optional arguments:
  -h, --help            show this help message and exit
  -v                   show invocation command with arguments
  -debug, --debug       invoke OCLint in debug mode
  -i INCLUDES, --include INCLUDES, --include INCLUDES
                        extract files matching pattern
  -e EXCLUDES, --exclude EXCLUDES, --exclude EXCLUDES
                        remove files matching pattern
```

3.2.1 Filter Options

-i INCLUDES, --include INCLUDES, --include INCLUDES Extract files matching pattern from `compile_commands.json` or prior matching result

-e EXCLUDES, --exclude EXCLUDES, --exclude EXCLUDES Remove files matching pattern from `compile_commands.json` or prior matching result

Sometimes, we may be interested in only a subset of entire codebase defined in `compile_commands.json`, and just want to inspect these sources. To do that, we can use filter options to get this subset. Since `oclint-json-compilation-database` is written in Python, so the matching pattern needs to follow [Python regular expression syntax](#). In addition, multiple filters can be chained to get the file set we need for analysis.

For example, if files like `Debug.m`, `Port.m` along with all tests files (saved under `Test` folder) need to be filtered out, try the following command or something similar:

```
oclint-json-compilation-database -e Debug.m -e Port.m -e Test
```

3.2.2 OCLint Options

Remember there are [many options](#) that we can use to change the behavior of OCLint itself? Sure, and we can also ask `oclint-json-compilation-database` to pass through these options when it invokes `oclint` under the hook.

Since all compiler options are defined in `compile_commands.json` file, we don't need to tell `oclint` about them this time. But by following the same idea, now, the OCLint options can be given to `oclint-json-compilation-database` by appending `-- separator` followed by all OCLint options:

```
oclint-json-compilation-database [<filter0> ... <filterN>] -- [oclint options]
```

3.2.3 Debug Options

-v show invocation command with arguments

-debug invoke OCLint in debug mode

`oclint-json-compilation-database` generates `oclint` command with corresponding options based on our settings of all filters and OCLint options, and eventually call `oclint` for us. Debug option `-v` outputs the final `oclint` invocation command, it's helpful for debugging purposes, because if we run the generated `oclint` command directly in the console, we will get the identical result same as using `oclint-json-compilation-database`.

More than that, if OCLint is built in the debug model, `-debug` outputs deeper message from OCLint invocation. It prints messages that can help understand the overall progress of OCLint analysis. Please aware that this is only available when OCLint is built with debug flag on.

3.3 oclint-xcodebuild Manual

For developers who use Xcode, `oclint-xcodebuild` is a helpful program that extracts adequate compiler options from `xcodebuild` execution log, converts them into JSON Compilation Database format, and save it into `compile_commands.json` file.

See also:

Read Apple's official [xcodebuild Manual Page](#) from Mac Developer Library. To understand how `oclint-xcodebuild` can be applied in your workflow, please move onto [Using OCLint with xcodebuild](#) document.

Please also consider using [xcpretty](#).

3.3.1 Running oclint-xcodebuild

If a `xcodebuild.log` file is at the current working directory. Simply run

```
oclint-xcodebuild
```

In case the `xcodebuild` log is stored in a file with name other than `xcodebuild.log`, append the path to the log file like

```
oclint-xcodebuild </path/to/xcodebuild/log>
```

When certain files are compiled by Xcode but we want to exclude them from the compilation database, we could filter them out by

```
oclint-xcodebuild -e <exclude_pattern_regex> </path/to/xcodebuild/log>
```

The `compile_commands.json` will be generated to the current working directory.

```
oclint-xcodebuild -o </path/to/output/compile/commands/json> </path/to/xcodebuild/log>
```

In addition, this will redirect the JSON Compilation Database output to the path specified.

Warning: It's highly recommended to avoid having spaces in your file name or file path. It's a good practice. Meanwhile, even though `oclint` itself supports spaces in path, `oclint-json-compilation-database` and `oclint-xcodebuild` still have issues handling spaces in path.

Note: `oclint-xcodebuild` is still an experimental project. The success of it depends on various things, e.g. macOS version, the Xcode version and project settings. However, since developers who use Xcode are familiar with Apple's manner of supporting only the latest version and one previous version, so `oclint-xcodebuild` tries to follow this convention. Your feedback is warmly welcome to help improve this helper program.

4.1 Using OCLint with CMake

This document shows how to apply OCLint to the projects which use CMake as build system.

4.1.1 Prerequisite

- [CMake](#)
- [oclint Manual](#)
- [oclint-json-compilation-database Manual](#)

4.1.2 Background

CMake is able to generate JSON Compilation Database file `compile_commands.json` with no hassle at all. If a project uses CMake as its build system, then applying OCLint is a quite easy task.

4.1.3 Generating `compile_commands.json`

Simply add `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` to the existing `cmake` command, like

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/source-root
```

As a result, the `compile_commands.json` file is generated in the CMake build folder.

That's it!

4.1.4 Using `compile_commands.json`

We can leave the `compile_commands.json` file here in the build directory, and use `-p` option to specify this file for `oclint`.

But in order to use `oclint-json-compilation-database`, it's required to copy or link this file to our source directory, for example:

- copy the file

```
cp `pwd`/compile_commands.json /path/to/source-root
```

- link the file

```
ln -s `pwd`/compile_commands.json /path/to/source-root
```

4.1.5 Running OCLint

Now we can use `oclint` by

```
oclint -p <build path> <source0> [... <sourceN>]
```

`build path` is the path to the folder that contains `compile_commands.json`.

In addition, we can use `oclint-json-compilation-database` for code analysis. In this case, simply enter

```
oclint-json-compilation-database
```

For advanced usage, detail instructions can be found in `oclint` and `oclint-json-compilation-database` manuals respectively.

4.2 Using OCLint with Bear

This document shows how to apply OCLint to the projects which use `Make` and other build systems in unix-like operating systems.

4.2.1 Prerequisite

- `Bear`
- `man make`
- `oclint` Manual
- `oclint-json-compilation-database` Manual

4.2.2 Background

Bear is a very handy tool to generate compilation database during the build process. Bear is a very important supplement especially for those who don't use CMake as build system. Bear can be applied in a wider circumstances, because it injects into the build process, and intercepts the `exec` calls to understand the compilation details.

4.2.3 Generating compile_commands.json

By following the instructions on [Bear README](#), we could have `bear` ready to use.

For example, if want to generate the `compile_command.json` for a project using `make`, we can easily use `bear` by

```
$ /path/to/bear make
```

4.2.4 What's Next

The rest of the process is as same as those who use CMake. Please refer to the [other document](#).

4.3 Using OCLint with xcodebuild

This document goes through the happy path of using OCLint to analyze the code quality of a Xcode project with `xcodebuild`.

See also:

If you use Facebook's `xctool` to build your Xcode projects, please use the [json-compilation-database](#) reporter to make things much easier. We highly recommend you start to use `xctool` as a replacement for `xcodebuild`.

In addition, please also consider using `xcpretty` if you use `xcodebuild`.

4.3.1 Prerequisite

- [oclint Manual](#)
- [oclint-json-compilation-database Manual](#)
- [oclint-xcodebuild Manual](#)
- Apple's official [xcodebuild Manual Page](#)

4.3.2 Background

OCLint recognizes a file called `compile_commands.json` to figure out the compiler options for parsing each file. For Xcode users, since all these compiler options are implicitly configured in Xcode build settings, we can see what actually happens when we invoke `xcodebuild` in terminal. Our approach is to capture the log of `xcodebuild` output, use `oclint-xcodebuild` to extract the adequate compiler options, convert them into JSON Compilation Database format, and save it into `compile_commands.json` file. Then we can use `oclint-json-compilation-database` to run analysis.

4.3.3 Running xcodebuild

Running `xcodebuild` is a quite simple task to some people by figuring out the correct options for `xcodebuild`. However, some people may feel it's not intuitive, so be patient, and take your time. You may find many online tutorials and blog posts that may help.

Basically, let's say we have a `DemoProject`, to know all the options we have, enter `xcodebuild -list`:

```
$ xcodebuild -list
Information about project "DemoProject":

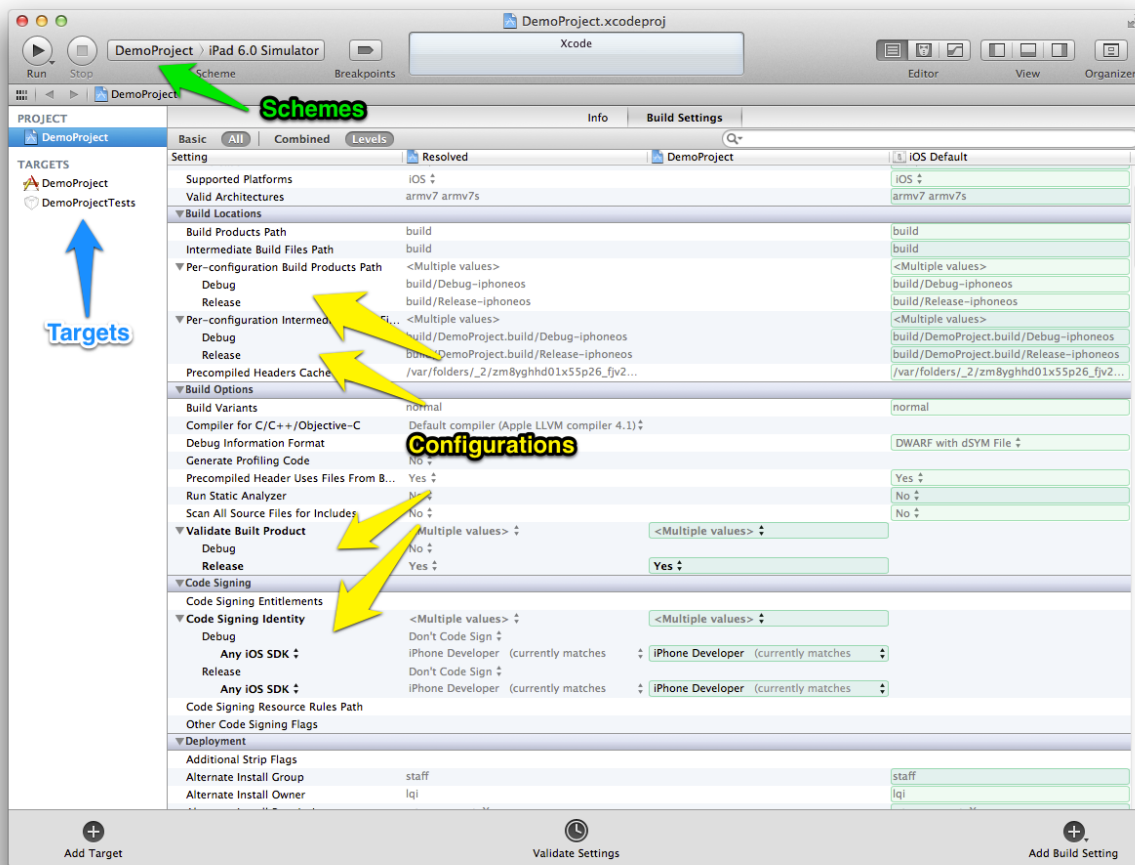
Targets:
    DemoProject
    DemoProjectTests

Build Configurations:
    Debug
    Release

    If no build configuration is specified and -scheme is not passed then "Release"
    is used.

Schemes:
    DemoProject
```

We can map these options back to the Xcode GUI:



Based on our choices in Xcode, we can set the corresponding options for `xcodebuild`. For instance, we can build our `DemoProject` project with

```
xcodebuild -target DemoProject -configuration Debug -scheme DemoProject
```

We should see detail `xcodebuild` invocations with a `** BUILD SUCCEEDED **` in the end. Great!

There are more options for `xcodebuild`, like `workspace`, `arch`, `sdk`, etc, we can apply them when necessary.

4.3.4 Capturing Output

We need to save the `xcodebuild` output to a log file, by convention, name it `xcodebuild.log`. We can use `xcodebuild <options> | tee xcodebuild.log` to pipe every line of the output to `xcodebuild.log` file.

4.3.5 Running oclint-xcodebuild

By running `oclint-xcodebuild` in the project root folder, a `compile_command.json` file should be generated.

4.3.6 Running oclint-json-compilation-database

Excellent! Last step, we can kick off code analysis by `oclint-json-compilation-database`. We can filter the JSON compilation database, and analyze on the files we are interested. We can also change the OCLint behaviors to meet our specific requirements.

4.3.7 Discussions

Clean Build

If a source file has been built by `xcodebuild`, and it's not been modified since last build, then it might not be compiled again when you invoke `xcodebuild` the second time. In other words, if it happens, this file won't be shown in the log. So we won't see it in the `compile_commands.json`. To avoid that, use clean build by removing all build products and intermediate files from the build directory.

However, cleaning and building the entire project takes longer time, especially for those big projects. In this case, if file structure hasn't been changed, and build settings haven't been modified, then it's okay to keep the existing `xcodebuild.log` and `compile_commands.json` to save time.

If the `xcodebuild` build can be guaranteed to be successful with the options specified, then we could also use `-dry-run` option to *build* the project without actually running the commands, so that we can still capture the `xcodebuild` log but with reduced time.

Save to Repository

`compile_commands.json` is platform sensitive. Whenever the environment has been changed, for example, upgrading Mac OS, upgrading Xcode version, switch to another SDK, and so on, please remove the existing `xcodebuild.log` and `compile_commands.json`, capture `xcodebuild` log again with the updated options, and re-generate new `compile_commands.json`.

Checking `compile_commands.json` into source code repository is not necessary. Instead, always generate a new `compile_commands.json` when anything changes.

However, we could write the entire process into a bash script, and check in this script. So that, all developers who work on the project can run this script and generate the `compile_commands.json` file that works best for his or her local environment.

See also:

You might also be interested in [using OCLint in Xcode](#).

4.4 Using OCLint with xctool

This document goes through the happy path of using OCLint to analyze the code quality of a Xcode project with xctool.

4.4.1 Prerequisite

- [oclint Manual](#)
- [oclint-json-compilation-database Manual](#)
- [xctool README](#)

4.4.2 Background

xctool is a drop-in replacement for Apple's xcodebuild. It makes building Xcode projects much easier. It eases continuous integration in many ways as well. xctool supports generating the report in JSON Compilation Database format, which can be used by OCLint to understand all the compiler flags in order to analyze the code.

4.4.3 Running xctool with json-compilation-database Reporter

Running xctool is quite straight forward. For example, the command below will build the project and generate the compile_commands.json file under the current folder.

```
path/to/xctool.sh \  
-project YourProject.xcodeproj \  
-scheme YourScheme \  
-reporter json-compilation-database:compile_commands.json \  
build
```

xctool also supports an .xctool-args file to preserve the command arguments, like workspace, scheme, configuration, etc. If we have .xctool-args ready, then we could simply run

```
xctool -reporter json-compilation-database:compile_commands.json build
```

to get the compile_commands.json file.

We can use xctool clean to clean the build in order to have a full list of all compiling files the next time we call xctool build.

4.4.4 Running oclint-json-compilation-database

Now, by having the compile_commands.json file, we can run the code analysis by simply call

```
oclint-json-compilation-database
```

Or with your customizations.

See also:

With the great help from xctool, you could even [show OCLint warnings in Xcode](#).

4.5 Using OCLint with xcpretty

This document goes through using xcpretty inside OCLint’s workflow for analyzing the code quality for a Xcode project.

4.5.1 Prerequisite

- [oclint Manual](#)
- [oclint-json-compilation-database Manual](#)
- [xcpretty README](#)

4.5.2 Generating json-compilation-database with xcpretty

Running xcpretty is quite straight forward. For example, the command below will build the project and generate the `compile_commands.json` file under the current folder.

```
xcodebuild [flags] | xcpretty -r json-compilation-database -o compile_commands.json
```

If you want to preserve the raw xcodebuild output, then you can do

```
xcodebuild [flags] | tee xcodebuild.log | xcpretty -r json-compilation-database -o ↪ compile_commands.json
```

4.5.3 Running oclint-json-compilation-database

Now, by having the `compile_commands.json` file, we can run the code analysis by simply call

```
oclint-json-compilation-database
```

Or with your customizations.

4.6 Using OCLint in Xcode

This document shows one solution of using OCLint to analyze the code quality of a Xcode project.

4.6.1 Prerequisite

- [oclint-xcodebuild Manual](#)
- [Apple’s official xcodebuild Manual Page](#)
- [Using OCLint with xcodebuild](#)

or

- [Using OCLint with xctool](#)

or

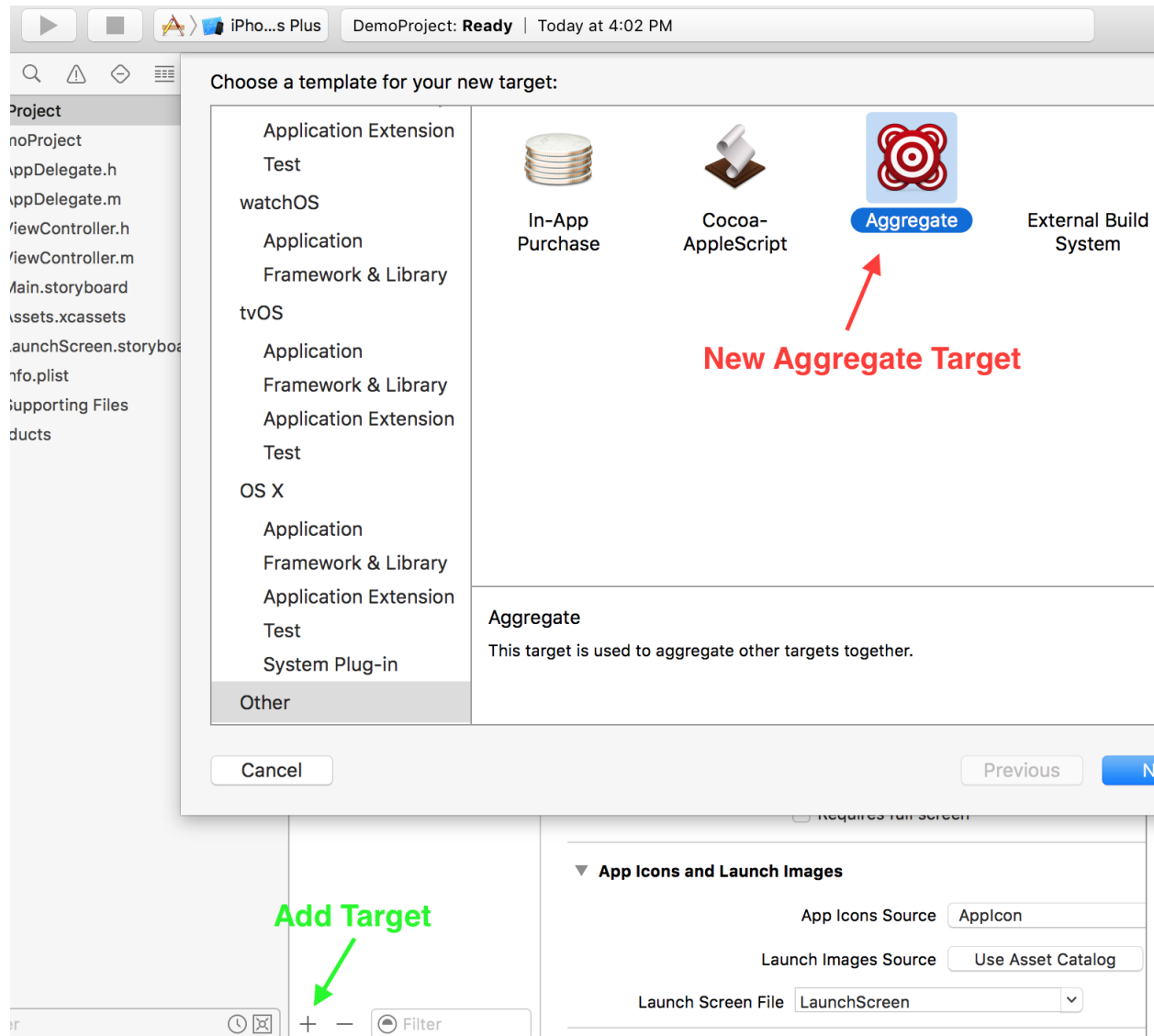
- [Using OCLint with xcpretty](#)

4.6.2 Background

This idea was originally posted in [this blog](#). We hope to share it with more developers, and hope to motivate more ideas.

4.6.3 Setting up Target

- Add a new target in the project, and choose `Aggregate` as the template.



- Name the new target, here we simply call it “OCLint”, you could have more than one targets that focus on different aspects of the code analysis.

Choose options for your new target:

Product Name:

Project:

Cancel

Previous

Finish

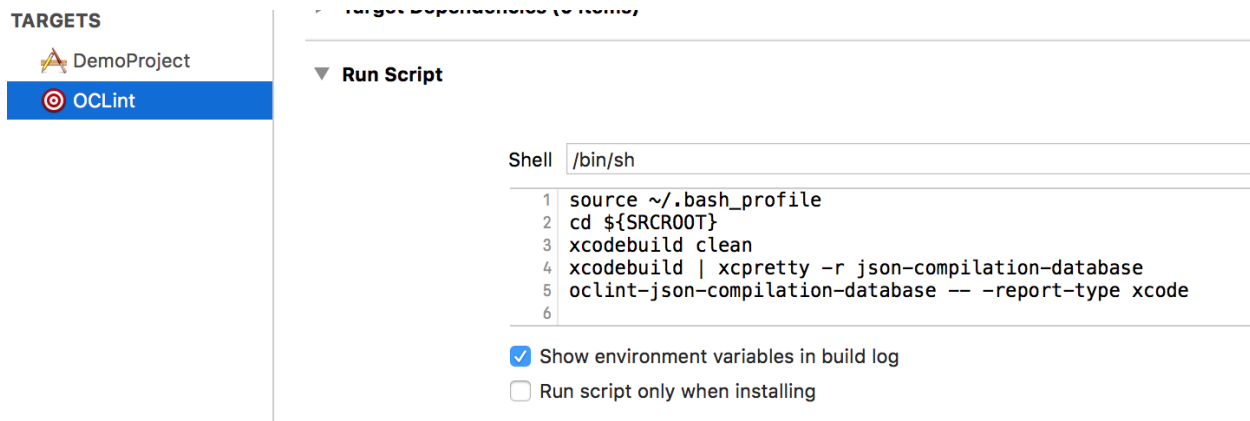
- Add a new build phase in the target we just created. Choose Add Run Script for the phase type.

The screenshot shows the Xcode interface with the 'Build Phases' tab selected. On the left, the 'PROJECT' list contains 'DemoProject' and 'OCLint' (selected). The 'TARGETS' list also contains 'DemoProject' and 'OCLint' (selected). In the main area, the 'Run Script' build phase is expanded, showing a '+' button to add a new script. A blue arrow points from the text 'Add Build Phase Choose "Add Run Script"' to this '+' button. The 'Run Script' section shows a shell of '/bin/sh' and a list of input files with the first item being 'Type a script or drag a script file to its path'. There are also checkboxes for 'Show environment variables in build log' (checked) and 'Run script only when installing' (unchecked).

- In the script editor, we could enter the script which does the real work. We can also modify the script from this

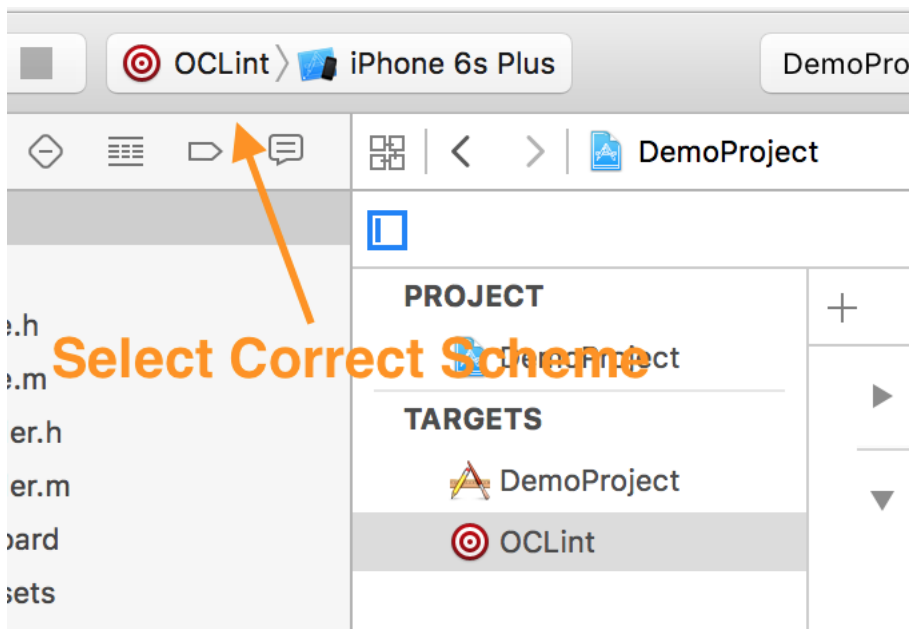
very [generic version](#) and [its folks](#). We may need to change the `xcodebuild` options to use a particular scheme or target. In addition, based on the [discussions](#) we had, we can decide whether to use `clean` and `dry run` features.

- For `xctool` users, the script can be largely simplified to something like [this](#).
- For `xcpretty` users, the script is also much simpler, check it out from [this gist](#).



4.6.4 Running Analysis

- Choose the correct build scheme, here we choose OCLint.



- Click to build, or use the shortcut `Command+B`.
- When the progress bar scrolls to the very right, the analysis is done, then we can check out the analysis results same as compile warnings.


```

30
31 - (NSString *)getValueForKey:(NSString *)key
32 {
33     NSData *valueData = [self getValueData];
34     if (valueData != nil)
35     {
36         NSString *value = [[NSString alloc] initWithData:valueData
37                               encoding:NSUTF8StringEncoding];
38         return value;
39     }
40     else
41     {
42         return nil;
43     }
44 }
45

```

Annotations in the original image:

- Line 31: Unused method parameter [unused|P3] The parameter 'key' is unused.
- Line 34: Inverted logic
- Line 41: Unnecessary else statement

4.7 Using OCLint with Travis CI

4.7.1 Objective-C/Xcode

The script you have for [running OCLint for Xcode](#) can be reused here.

```

language: objective-c
osx_image: xcode7.2
before_install:
- brew cask uninstall oclint
- brew tap oclint/formulae
- brew install oclint
script:
- xcodebuild | tee xcodebuild.log
- oclint-xcodebuild
- oclint-json-compilation-database

```

An example can be found at this [github repository](#).

4.7.2 C++/CMake

OCLint is pre-installed on macOS images, however, not on linux images, so we provide a script to automate the installation and set it up for you. Feel free to review the [contents of the script](#) before using it.

```

os:
- linux
- osx
language: cpp
sudo: required
dist: trusty
osx_image: xcode7.2
before_install:
- if [ $TRAVIS_OS_NAME == osx ]; then brew update && brew install cmake; fi
- if [ $TRAVIS_OS_NAME == linux ]; then eval "$(curl -sL https://raw.
  ↳githubusercontent.com/ryuichis/oclint-cpp-travis-ci-examples/master/oclint-ci-
  ↳install.sh)"; fi
script:
- mkdir build
- cd build

```

(continues on next page)

(continued from previous page)

```
- cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..
- cd ..
- cp build/compile_commands.json .
- oclint-json-compilation-database
```

An example can be found at this [github repository](#).

4.8 Using OCLint with Jenkins CI

This document shows how to integrate OCLint into Jenkins CI, and render the graphic report by the cooperation between Jenkins' PMD Plugin and OCLint's PMD reporter.

4.8.1 Prerequisite

- [Jenkins CI](#)
- [PMD Plugin for Jenkins CI](#)
- Knowing how to configure Jenkins to run [continuous integration](#) for the project
- Knowing how to generate `compile_commands.json` for the project
 - Read other guides in this chapter for references
- [oclint Manual](#)
- [oclint-json-compilation-database Manual](#)

4.8.2 Background

From the very beginning of OCLint project, we have always taken continuous integration into serious consideration. OCLint facilitates this process a lot. It generates the certain type of report as artifact, and it fails your build if any level of violations exceeds the threshold.

In order to utilize the Jenkins CI, we have introduced the PMD reporter that converts the analysis results into the format very close to PMD reports, so that the PMD plugin inside Jenkins can easily picks up this output, categorizes and renders graphic reports for us.

The idea was originally developed by Stephan Esch in his [blog post](#) along with his contribution of writing `PMDReporter` for us.

4.8.3 Setting up Project

- Create a new free-style project

Jenkins search

Jenkins All

Job name DemoProject

New Job

Build a free-style software project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Build a maven2/3 project
Build a maven 2/3 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Build multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Monitor an external job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

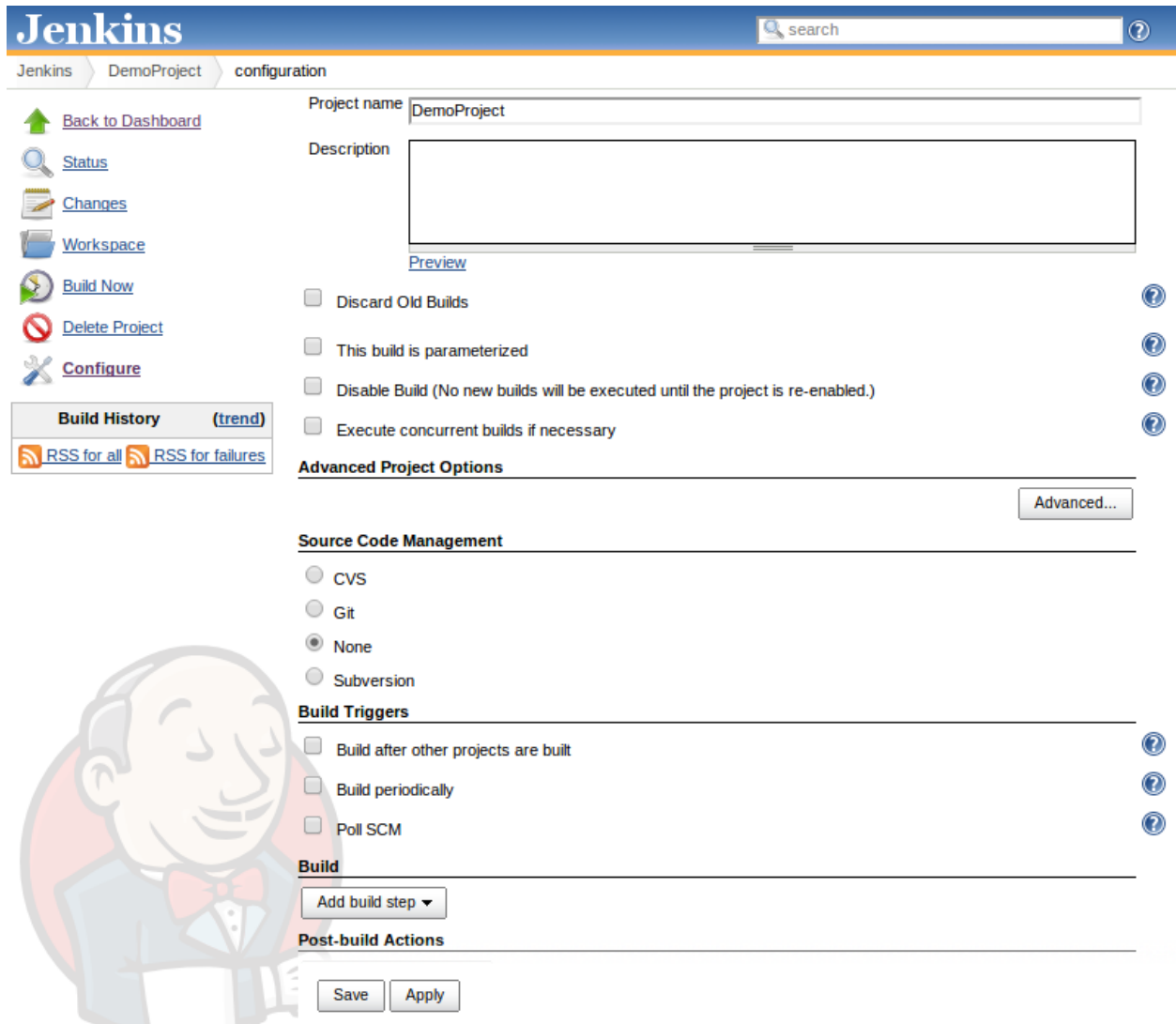
Build Queue
No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

OK

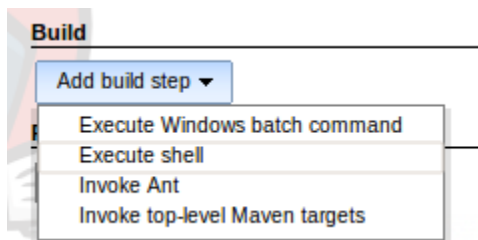
- Set up adequate steps for continuously building the project



The screenshot shows the Jenkins web interface for configuring a project named 'DemoProject'. The top navigation bar includes 'Jenkins', 'DemoProject', and 'configuration'. A sidebar on the left contains links: 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', 'Configure', 'Build History (trend)', and 'RSS for all'/'RSS for failures'. The main configuration area includes a 'Project name' field with 'DemoProject', a 'Description' text area, and a 'Preview' button. Below these are several checkboxes: 'Discard Old Builds', 'This build is parameterized', 'Disable Build (No new builds will be executed until the project is re-enabled.)', and 'Execute concurrent builds if necessary'. A section titled 'Advanced Project Options' has an 'Advanced...' button. Under 'Source Code Management', radio buttons are selected for 'None' (others are CVS, Git, Subversion). Under 'Build Triggers', checkboxes for 'Build after other projects are built', 'Build periodically', and 'Poll SCM' are shown. The 'Build' section has an 'Add build step' button. The 'Post-build Actions' section is empty. At the bottom are 'Save' and 'Apply' buttons. A large, faint watermark of a man in a tuxedo is visible on the left side of the configuration area.

4.8.4 Configuring OCLint and PMD Plugin

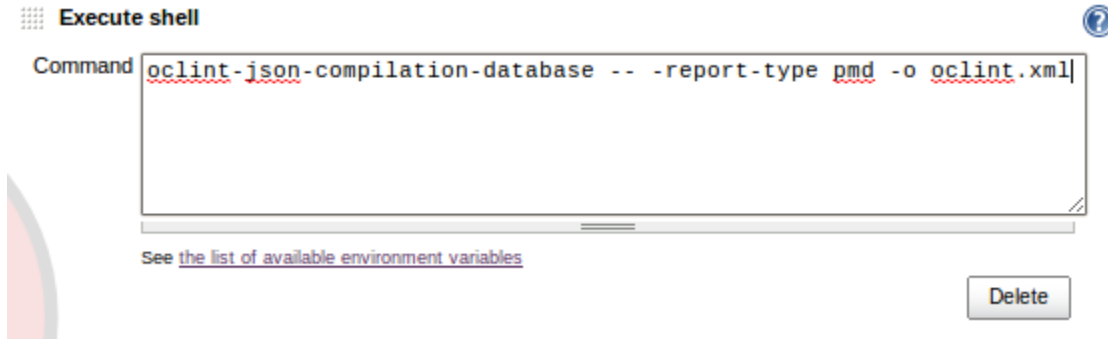
- Add a new build step as an execute shell



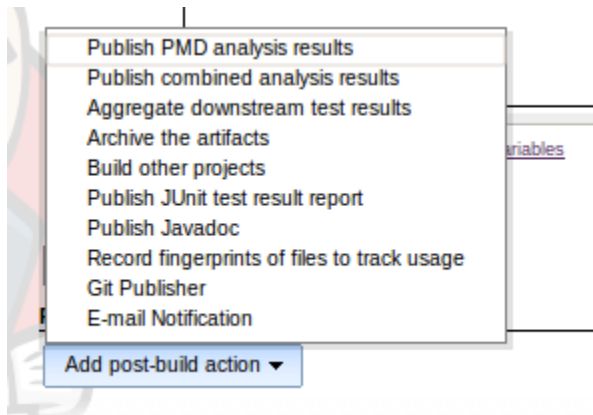
This screenshot shows the 'Add build step' dropdown menu from the Jenkins configuration page. The menu is open, displaying four options: 'Execute Windows batch command', 'Execute shell', 'Invoke Ant', and 'Invoke top-level Maven targets'. The 'Build' section header is visible above the dropdown.

- Set up command used for OCLint analysis
 - In addition to the script shown in the following screenshot, we may also need to write the `compile_commands.json` generation commands in advance.
 - In some cases, `oclint` binary suits better than `oclint-json-compilation-database`

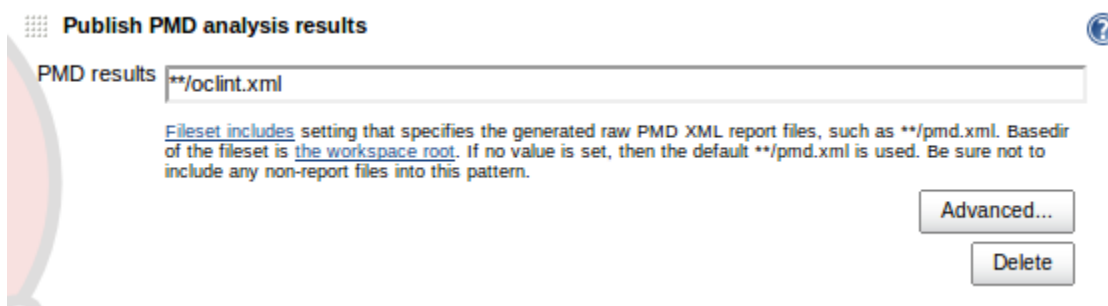
- We need to set the report type to pmd
- Give a name to the output artifact, we will need this file in the following step



- Add a new post-build action, and choose Publish PMD analysis results

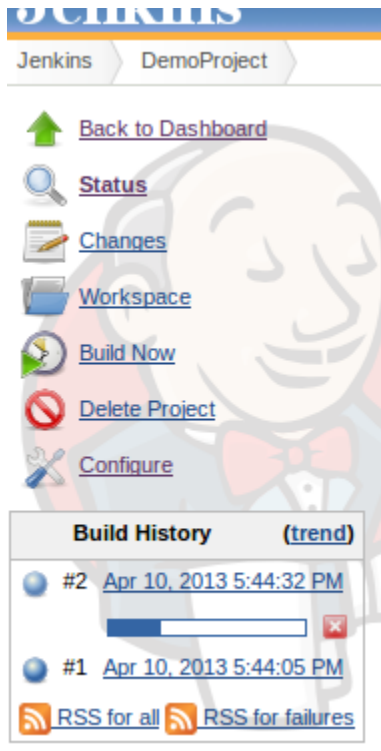


- Enter the output artifact name that matches previous step



4.8.5 Running Analysis

Same as running regular Jenkins build, it might be triggered by time, by pinging the source control management, and manually. you can see the build history and current build progress.



4.8.6 Monitoring Results

When the build finishes successfully, we now can go into that build. On the left sidebar, click `PMD Warnings` to see the report. Following is a screenshot of sorting the results by the type of the rules.

Jenkins DemoProject #2 PMD Warnings

[Back to Project](#)[Status](#)[Changes](#)[Console Output](#)[Edit Build Information](#)[Delete Build](#)[Git Build Data](#)[No Tags](#)**PMD Warnings**[Previous Build](#)

PMD Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
17	0	0

Summary

Total	High Priority	Normal Priority	Low Priority
17	0	7	10

Details

Files	Types	Warnings	Details	Normal	Low
Type	Total	Distribution			
collapsible if statements	1	<div></div>			
dead code	1	<div></div>			
empty for statement	1	<div></div>			
empty while statement	1	<div></div>			
high cyclomatic complexity	2	<div></div>			
high ncss method	1	<div></div>			
long line	1	<div></div>			
long method	1	<div></div>			
missing break in switch statement	2	<div></div>			
parameter reassignment	2	<div></div>			
switch statements should have default	1	<div></div>			
too few branches in switch statement	2	<div></div>			
unused method parameter	1	<div></div>			
Total	17				

5.1 Customize Rule Thresholds

Rule system in OCLint is designed to be very flexible and dynamic for various development cultures in many ways. Particularly, this document will go through customizing the behaviors of the rules by changing the thresholds.

Certain rules emit violations only when the metrics of the source code exceed the thresholds. For example, according to [McCabe76], a reasonable cyclomatic complexity number for a method should be less than 10. So, default threshold in CyclomaticComplexityRule is set to 10. However, this value may not be the best for a very project. For instance, a more sophisticated legacy codebase comes with high complexity, or on the other side, one team may in the middle of pushing a much more strict coding standard. For these cases, customizations can be achieved by changing thresholds.

5.1.1 Command Line Usages

Customizing thresholds by appending `-rc <threshold_name>=<new_value>` option to the command line invocation. Multiple “-rc”s are accepted for changing more than one thresholds. When multiple values are given to the same threshold, the last one will be taken and overrides the default or existing custom values in the configuration files. For example, in order to change cyclomatic complexity number to 15, but to lower the long line to 50, following command can be given:

```
-rc CYCLOMATIC_COMPLEXITY=15 -rc LONG_LINE=50
```

5.1.2 Configuration File

When the entire team shares the same standard, it's recommended to define the thresholds in a configuration file, and check in the file into source code system. So that OCLint behaves consistently within the team. The section for changing thresholds is:

```
rule-configurations:  
  - key: THRESHOLD_1
```

(continues on next page)

(continued from previous page)

```

    value: new_value_1
- key: THRESHOLD_2
    value: new_value_2
...

```

For the same example, changing cyclomatic complexity number to 15 and lowering the long line threshold to 50 can also be achieved by putting the following into project's `.oclint` file:

```

rule-configurations:
- key: CYCLOMATIC_COMPLEXITY
  value: 15
- key: LONG_LINE
  value: 50

```

5.1.3 Available Thresholds

Here is a list of all thresholds defined in OCLint 20.11:

Name	Description	De- fault
CYCLO-MATIC_COMPLEXITY	Cyclomatic complexity of a method	10
LONG_CLASS	Number of lines for a C class or Objective-C interface, category, protocol, and implementation	1000
LONG_LINE	Number of characters for one line of code	100
LONG_METHOD	Number of lines for a method or function	50
LONG_VARIABLE_NAME	Number of characters for a variable name	20
MAXIMUM_IF_LENGTH	Number of lines for the <code>if</code> block that would prefer an early exists	15
MINI-MUM_CASES_IN_SWITCH	Count of case statements in a switch statement	3
NPATH_COMPLEXITY	NPath complexity of a method	200
NCSS_METHOD	Number of non-commenting source statements of a method	30
NESTED_BLOCK_DEPTH	Depth of a block or compound statement	5
SHORT_VARIABLE_NAME	Number of characters for a variable name	3
TOO_MANY_FIELDS	Number of fields of a class	20
TOO_MANY_METHODS	Number of methods of a class	30
TOO_MANY_PARAMETERS	Number of parameters of a method	10

See also:

OCLint configuration file Documentation for OCLint configuration file

Rule Selection Documentation of picking up the right rules for analysis

Write Own Rules Documentation of modifying existing rule code or even write own rules to gain more capabilities.

5.2 Select OCLint Rules for Inspection

OCLint is a rule based code analysis tool. Rule system is designed to be very extensible and flexible. So it's easy to customize rules in many ways, like making particular rule set for certain projects, categorize them and load them from different locations.

By default, rules will be loaded from `$(/path/to/bin/oclint)/../lib/oclint/rules` directory, we name it **rule search path** or **rule loading path**. Rule search path consists of a group of dynamic libraries, with extension `.so` in Linux, `.dylib` in macOS, and `.dll` in Windows. These dynamic libraries can be easily re-arranged or selected into new rule set for a particular project. This *plug-and-use* rule loading mechanism - all dynamic libraries are loaded when OCLint searches through rule loading paths - makes the tool very dynamic. Dragging and dropping new rules into the rule loading paths makes them immediate available for use. And multiple rule search paths are acceptable for one project, different rule loading paths can be specified for different projects.

In addition to building own rule set, rules can be picked or filtered out from an existing rule set, like the default one shipped with OCLint release bundle. This can be achieved through command line invocation or saving as configuration file.

5.2.1 Command Line Usages

Rule search path can be switched with `-R <directory>` option. When multiple locations can be specified, and rules in all these paths will be loaded.

Selecting rules from the rule search path by `-rule <rule name>`. On the other side, filtering rules out by `-disable-rule <rule name>`.

For example, when all rules except `GotoStatement` from rule search path `/path/to/rules` are applied in the analysis, following command can be used:

```
oclint -R /path/to/rules -disable-rule GotoStatement
```

5.2.2 Configuration File

Rule selections can be preserved into configuration files, and check in to source code system for team sharing the same standard. The same example for configuration file:

```
rule-paths:
- /path/to/rules
rules:
disable-rules:
- GotoStatement
```

See also:

OCLint configuration file Documentation for OCLint configuration file

Write Own Rules Documentation of modifying existing rule code or even write own rules to gain more capabilities.

5.3 Pick Up the Right Reporter

OCLint currently supports generating multiple types of reports based on the demands of different use cases. The generation for each report type is handled by a reporter. All reporters are collected in `oclint-reporters` module.

5.3.1 Report Options

In order to use other reporter, add `-report-type <report name>` to `oclint` command. OCLint uses plain text reporter by default, so usually `-report-type text` flag is not necessary unless other report type has been specified in configuration file and needs to be overridden.

Some reports are better to be viewed in places other than the console. For example, web browser is a good place for reading HTML reports, and PMD report has a better visual rendering effect in continuous integration systems like Jenkins. In these cases, `-o <path>` option can help redirect the report to a file instead of console.

By combining both options, for example, outputting the result in HTML format to `oclint_result.html` file can be achieved by

```
oclint -report-type html -o oclint_result.html <sources> -- <compiler flags>
```

5.3.2 Report Types

The following reporters are available. The names in the parentheses are the values used for `report-type` option.

Plain Text Report (text)

Plain text report is designed for directly outputting results to console.

Report starts with the summary of the inspection, it consists of number of total files, number of files with violations, and number of violations for three different priorities.

The output of each violation is formatted similar to compiler errors. Each line in the report indicates a violation. It starts with the file name, line, and column of the source code that the violation occurs. The name of the violated rule is printed after, followed by the priority of the rule. OCLint also outputs descriptions if current violation has any.

[Sample text report](#)

HTML Report (html)

HTML reporter is browser friendly with better readability. The entire report follows the same order as the text report, but with a nicer structure of sections, paragraphs, and annotations. Violations are highlighted with different colors according to the priority.

[Sample HTML report](#)

XML Report (xml)

XML reporter produces an XML report of the results.

[Sample XML report](#)

JSON Reporter (json)

JSON reporter produces an JSON report of the results.

[Sample JSON report](#)

PMD Reporter (pmd)

Since [PMD](#) report is supported by many existing continuous integration (CI) for Java developers, PMD reporter outputs the XML report that follows the PMD report format. So that these CI systems can pick up the output and render better graphic results.

[Sample PMD report](#)

Xcode Reporter (xcode)

Xcode reporter can be used inside Xcode IDE.

See also:

Read [this document](#) for using OCLint in Xcode.

Details of using OCLint

5.3.3 Configuration File

When a type of reporter is selected by the entire team, it's recommended to save it into `.oclint` file. For example, producing HTML report can be configured as:

```
report-type: html
output: oclint.html
```

See also:

[Write Own Reporters](#) Documentation of writing own reporters to extend OCLint with more capabilities.

5.4 Suppress OCLint Warnings

There are two scenarios you might want to suppress an warning:

- the standard for certain code measurement is controversial
- it is a false positive

There are several methods you could consider to suppress OCLint rule violations:

- If the warning you need to suppress also appeals to other OCLint users, e.g. a false positive
 - Patch test cases, amend the rule to make tests pass, and submit a pull request. Your help would be greatly appreciated by everyone
 - Open an issue with your code snippet on github or discuss with us in the mailing list
- If the standard of your project is slightly different from OCLint default settings
 - Consider [change rules' thresholds](#) to best fit the current environment
- If certain rules are really annoying for your projects
 - [Remove them from the rule search path](#)
 - [Add these rules into blacklist](#)
- If it is a case-by-case thing, and only particular places need to be suppressed
 - [Use annotations](#)
 - [Use !OCLint comment](#)

5.4.1 Annotations

By adding annotations to the source code, you can communicate with OCLint and order it to discard certain types of rule violations within a context-based range.

Annotation Syntax

You can suppress one rule like this:

```
__attribute__((annotate("oclint:suppress[unused method parameter]")))
```

Or, you can suppress multiple rules at the same time, like this:

```
__attribute__((annotate("oclint:suppress[high cyclomatic complexity]"), annotate(
↪ "oclint:suppress[high npath complexity]"), annotate("oclint:suppress[high ncss_
↪ method]")))
```

OCLint also allows you to suppress all rules with a shortcut:

```
__attribute__((annotate("oclint:suppress")))
```

5.4.2 Effective Range

Annotations are added to a particular declaration of the source code, and each suppress has the impact on the same scope of the declaration, usually this scope covers the current declaration itself and all its children.

For example, if an annotation is added to a method declaration, then the suppress effects the method, its entire content, including all its parameters, statements, expression, local variables, and so on.

Examples

```
bool __attribute__((annotate("oclint:suppress"))) aMethod(int aParameter)
{
    // warnings within this method are suppressed at all
    // like unused aParameter variable and empty if statement
    if (1) {}

    return true;
}

- (IBAction)turnoverValueChanged:
    (id) __attribute__((annotate("oclint:suppress[unused method parameter]"))) sender
    // suppress sender variable
{
    int i; // won't suppress this one
    [self calculateTurnover];
}

- (void)dismissAllViews:(id)currentView parentView:(id)parentView
    __attribute__((annotate("oclint:suppress")))
    // again, suppress the warnings for entire method
{
    [self dismissTurnoverView];
    // plus 30+ lines of code of dismissing other views
}
```

5.4.3 !OCLint Comment

Alternatively, the warnings of a specific line can be suppressed by appending `//!OCLint` comment, for example:

```
void a() {
    int unusedLocalVariable; //!OCLINT
}
```

As a good practice, it's highly recommend to add the reasoning for the comment, like:

```
int Results::numberOfWarnings()
{
    std::lock_guard<std::mutex> lock(_mutex); //!OCLint(FP - meant to be unused)
    //Everything after the letter 't' is ignored, but just for better readability
    return _compilerWarningSet->numberOfViolations();
}
```

Note that comment-based suppress doesn't care the violation type nor number of violations, it simply ignore every warnings on that specific line. So the comment is expected to be on the same line as the violation. For example, when empty if statement is supposed to be suppressed, `//!OCLint` needs to be put on the line containing if statement, e.g.:

```
if (true) //!OCLint goes here
{
    // it is empty
}
```

5.5 Preserve Configurations

OCLint can load configuration files (`/etc/oclint`, `~/.oclint` and `.oclint`) and populates arguments from there.

There are three levels of configurations files-

System Configuration File is saved into `$(/path/to/bin/oclint)/../etc/oclint`, when `oclint` main binary is installed into `/bin/oclint` for the most common case, then this is the regular `/etc/oclint` file. The configurations saved into this file change the behaviors of OCLint throughout the entire system.

User Configuration File is preserved to `~/.oclint`, which is under the home directory of the current user. User preferences can be saved into this file, and only scope the current user. Configurations in this file can override the existing ones in the system configuration file.

Project Configuration File is usually placed at the root folder of the project, where `oclint` is usually invoked from. This file is scanned and configurations are only for the current project. Configurations in this file can override the ones in user configuration file, and in sequence, override the ones in system configuration file.

Arguments passing to the command line invocation would override the ones in the configuration file.

The acceptable configuration file is written in YAML format, with the following available options:

Option	Type	Mapping Command Option
rules	List of strings	-rule
disable-rules	List of strings	-disable-rule
rule-paths	List of strings	-R
rule-configurations	List of associative arrays	-rc
output	String	-o
report-type	String	-report-type
max-priority-1	Integer	-max-priority-1
max-priority-2	Integer	-max-priority-2
max-priority-3	Integer	-max-priority-3
enable-global-analysis	Boolean	-enable-global-analysis
enable-clang-static-analyzer	Boolean	-enable-clang-static-analyzer

An example of a `.oclint` file might look like this:

```
disable-rules:
- LongLine
rulePaths:
- /etc/rules
rule-configurations:
- key: CYCLOMATIC_COMPLEXITY
  value: 15
- key: NPATH_COMPLEXITY
  value: 300
output: oclint.xml
report-type: xml
max-priority-1: 20
max-priority-2: 40
max-priority-3: 60
enable-clang-static-analyzer: false
```

See also:

OCLint manual Manual for OCLint main binary

Customizing Thresholds Documentation of changing the thresholds in order to modify the behavior of the tool

Rule Selection Documentation of picking up the right rules for analysis

Reporter Selection Documentation of selecting the reporters for output

5.6 Set Up OCLint in MinGW Environment

This page presents our effort of porting OCLint to MinGW environment.

Note: Certain features are not fully tested with MinGW environment, and your comments are welcome.

5.6.1 System Requirements

1. MinGW
2. MSYS

3. GCC and G++ Toolchain
4. Python
5. Git
6. Apache Subversion
7. CMake

5.6.2 Cleaning Up

Cleaning by making sure the build folder, llvm folder, googletest folder and oclint-json-compilation-database folder are deleted.

5.6.3 Release Build

The release build is produced by:

```
./ci -setup -release
```

This is actually a wrapper task consisting of:

```
cd ..
git clone https://github.com/oclint/oclint-json-compilation-database
# check out oclint-json-compilation-database code
cd oclint-scripts
./clang co          # check out LLVM/Clang code
./clang build -release # build Clang
./build -release     # build OCLint
./bundle            # bundle everything and be ready to use
```

5.6.4 Running Tests

Tests can tell if OCLint is stable on MinGW environment:

```
./googleTest co      # check out GoogleTest code
./googleTest build   # build GoogleTest
./test               # test all modules
```

5.6.5 Keeping Up To Date

Update OCLint itself by `git pull`, and same thing for `oclint-json-compilation-database` module.

Update Clang and GoogleTest by `./clang update` and `./googleTest update` respectively.

5.6.6 Verifying the Build

By calling the binary

```
./build/oclint-<major>.<minor>.dev.<git-hash>/bin/oclint.exe
```

The following error message is expected:

```
oclint: Not enough positional command line arguments specified!  
Must specify at least 1 positional arguments: See: ./oclint -help
```

See also:

OCLint manual Manual for OCLint main binary

Building OCLint in Unix-Like Environment Documentation of building OCLint on Linux and macOS

Testing OCLint in Unix-Like Environment Documentation of compiling and testing OCLint on Linux and macOS

OCLint 20.11 includes 71 rules.

6.1 Basic

6.1.1 BitwiseOperatorInConditional

Since: 0.6

Name: bitwise operator in conditional

Checks for bitwise operations in conditionals. Although being written on purpose in some rare cases, bitwise operations are considered to be too “smart”. Smart code is not easy to understand.

This rule is defined by the following class: [oclint-rules/rules/basic/BitwiseOperatorInConditionalRule.cpp](#)

Example:

```
void example(int a, int b)
{
    if (a | b)
    {
    }
    if (a & b)
    {
    }
}
```

6.1.2 BrokenNullCheck

Since: 0.7

Name: broken null check

The broken null check itself will crash the program.

This rule is defined by the following class: [oclint-rules/rules/basic/BrokenNullCheckRule.cpp](#)

Example:

```
void m(A *a, B *b)
{
    if (a != NULL || a->bar(b))
    {
    }

    if (a == NULL && a->bar(b))
    {
    }
}
```

6.1.3 BrokenNilCheck

Since: 0.7

Name: broken nil check

The broken nil check in Objective-C in some cases returns just the opposite result.

This rule is defined by the following class: [oclint-rules/rules/basic/BrokenNullCheckRule.cpp](#)

Example:

```
+ (void)compare: (A *) obj1 withObject: (A *) obj2
{
    if (obj1 || [obj1 isEqualTo:obj2])
    {
    }

    if (!obj1 && ![obj1 isEqualTo:obj2])
    {
    }
}
```

6.1.4 BrokenOddnessCheck

Since: 0.6

Name: broken oddness check

Checking oddness by `x % 2 == 1` won't work for negative numbers. Use `x & 1 == 1`, or `x % 2 != 0` instead.

This rule is defined by the following class: [oclint-rules/rules/basic/BrokenOddnessCheckRule.cpp](#)

Example:

```
void example()
{
    if (x % 2 == 1)           // violation
    {
    }
}
```

(continues on next page)

(continued from previous page)

```

    if (foo() % 2 == 1)    // violation
    {
    }
}

```

6.1.5 CollapsibleIfStatements

Since: 0.6

Name: collapsible if statements

This rule detects instances where the conditions of two consecutive if statements can be combined into one in order to increase code cleanness and readability.

This rule is defined by the following class: `oclint-rules/rules/basic/CollapsibleIfStatementsRule.cpp`

Example:

```

void example(bool x, bool y)
{
    if (x)                // these two if statements can be
    {
        if (y)            // combined to if (x && y)
        {
            foo();
        }
    }
}

```

6.1.6 ConstantConditionalOperator

Since: 0.6

Name: constant conditional operator

conditional operator whose conditionals are always true or always false are confusing.

This rule is defined by the following class: `oclint-rules/rules/basic/ConstantConditionalOperatorRule.cpp`

Example:

```

void example()
{
    int a = 1 == 1 ? 1 : 0;    // 1 == 1 is actually always true
}

```

6.1.7 ConstantIfExpression

Since: 0.2

Name: constant if expression

if statements whose conditionals are always true or always false are confusing.

This rule is defined by the following class: `oclint-rules/rules/basic/ConstantIfExpressionRule.cpp`

Example:

```
void example()
{
    if (true)           // always true
    {
        foo();
    }
    if (1 == 0)         // always false
    {
        bar();
    }
}
```

6.1.8 DeadCode

Since: 0.4

Name: dead code

Code after `return`, `break`, `continue`, and `throw` statements is unreachable and will never be executed.

This rule is defined by the following class: `oclint-rules/rules/basic/DeadCodeRule.cpp`

Example:

```
void example(id collection)
{
    for (id it in collection)
    {
        continue;
        int i1;           // dead code
    }
    return;
    int i2;               // dead code
}
```

6.1.9 DoubleNegative

Since: 0.6

Name: double negative

There is no point in using a double negative, it is always positive.

This rule is defined by the following class: `oclint-rules/rules/basic/DoubleNegativeRule.cpp`

Example:

```
void example()
{
    int b1 = !!1;
    int b2 = ~~1;
}
```

6.1.10 ForLoopShouldBeWhileLoop

Since: 0.6

Name: for loop should be while loop

Under certain circumstances, some `for` loops can be simplified to while loops to make code more concise.

This rule is defined by the following class: `oclint-rules/rules/basic/ForLoopShouldBeWhileLoopRule.cpp`

Example:

```
void example(int a)
{
    for (; a < 100;)
    {
        foo(a);
    }
}
```

6.1.11 GotoStatement

Since: 0.6

Name: goto statement

“Go To Statement Considered Harmful”

This rule is defined by the following class: `oclint-rules/rules/basic/GotoStatementRule.cpp`

Example:

```
void example()
{
    A:
        a();
    goto A;    // Considered Harmful
}
```

References:

Edsger Dijkstra (March 1968). “Go To Statement Considered Harmful”. *Communications of the ACM* (PDF) 11 (3): 147–148. doi:10.1145/362929.362947.

6.1.12 JumbledIncrementer

Since: 0.7

Name: jumbled incrementer

Jumbled incrementers are usually typos. If it’s done on purpose, it’s very confusing for code readers.

This rule is defined by the following class: `oclint-rules/rules/basic/JumbledIncrementerRule.cpp`

Example:

```
void aMethod(int a) {
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < a; i++) { // references both 'i' and 'j'
        }
    }
}
```

6.1.13 MisplacedNullCheck

Since: 0.7

Name: misplaced null check

The null check is misplaced. In C and C++, sending a message to a null pointer could crash the program. When null is misplaced, either the check is useless or it's incorrect.

This rule is defined by the following class: [oclint-rules/rules/basic/MisplacedNullCheckRule.cpp](#)

Example:

```
void m(A *a, B *b)
{
    if (a->bar(b) && a != NULL) // violation
    {
    }

    if (a->bar(b) || !a)         // violation
    {
    }
}
```

6.1.14 MisplacedNilCheck

Since: 0.7

Name: misplaced nil check

The nil check is misplaced. In Objective-C, sending a message to a nil pointer simply does nothing. But code readers may be confused about the misplaced nil check.

This rule is defined by the following class: [oclint-rules/rules/basic/MisplacedNullCheckRule.cpp](#)

Example:

```
+ (void)compare: (A *)obj1 withOther: (A *)obj2
{
    if ([obj1 isEqualTo:obj2] && obj1)
    {
    }

    if (![obj1 isEqualTo:obj2] || obj1 == nil)
    {
    }
}
```

6.1.15 MultipleUnaryOperator

Since: 0.6

Name: multiple unary operator

Multiple unary operator can always be confusing and should be simplified.

This rule is defined by the following class: [oclint-rules/rules/basic/MultipleUnaryOperatorRule.cpp](#)

Example:


```
void example()
{
    int b = -(+(! (~1)));
}
```

6.1.16 ReturnFromFinallyBlock

Since: 0.6

Name: return from finally block

Returning from a `finally` block is not recommended.

This rule is defined by the following class: `oclint-rules/rules/basic/ReturnFromFinallyBlockRule.cpp`

Example:

```
void example()
{
    @try
    {
        foo();
    }
    @catch(id ex)
    {
        bar();
    }
    @finally
    {
        return;           // this can discard exceptions.
    }
}
```

6.1.17 ThrowExceptionFromFinallyBlock

Since: 0.6

Name: throw exception from finally block

Throwing exceptions within a `finally` block may mask other exceptions or code defects.

This rule is defined by the following class: `oclint-rules/rules/basic/ThrowExceptionFromFinallyBlockRule.cpp`

Example:

```
void example()
{
    @try {};
    @catch(id ex) {};
    @finally {
        id ex1;
        @throw ex1;           // this throws an exception
        NSException *ex2 = [NSException new];
        [ex2 raise];          // this throws an exception, too
    }
}
```

6.2 Cocoa

6.2.1 MissingHashMethod

Since: 0.8

Name: missing hash method

When isEqual method is overridden, hash method must be overridden, too.

This rule is defined by the following class: `oclint-rules/rules/cocoa/ObjCVerifyIsEqualHashRule.cpp`

Example:

```
@implementation BaseObject

- (BOOL)isEqual:(id) obj {
    return YES;
}

/*
- (int)hash is missing; If you override isEqual you must override hash too.
*/

@end
```

6.2.2 MissingCallToBaseMethod

Since: 0.8

Name: missing call to base method

When a method is declared with `__attribute__((annotate("oclint:enforce[base method]")))` annotation, all of its implementations (including its own and its sub classes) must call the method implementation in super class.

This rule is defined by the following class: `oclint-rules/rules/cocoa/ObjCVerifyMustCallSuperRule.cpp`

Example:

```
@interface UIView (OCLintStaticChecks)
- (void)layoutSubviews __attribute__((annotate("oclint:enforce[base method]")));
@end

@interface CustomView : UIView
@end

@implementation CustomView

- (void)layoutSubviews {
    // [super layoutSubviews]; is enforced here
}

@end
```

6.2.3 CallingProhibitedMethod

Since: 0.10.1

Name: calling prohibited method

When a method is declared with `__attribute__((annotate("oclint:enforce[prohibited method]")))` annotation, all of its usages will be prohibited.

This rule is defined by the following class: `oclint-rules/rules/cocoa/ObjCVerifyProhibitedCallRule.cpp`

Example:

```
@interface A : NSObject
- (void)foo __attribute__((annotate("oclint:enforce[prohibited method]")));
@end

@implementation A
- (void)foo {
}
- (void)bar {
    [self foo]; // calling method `foo` is prohibited.
}
@end
```

6.2.4 CallingProtectedMethod

Since: 0.8

Name: calling protected method

Even though there is no `protected` in Objective-C language level, in a design's perspective, we sometimes hope to enforce a method only be used inside the class itself or by its subclass. This rule mimics the `protected` behavior, and alerts developers when a method is called outside its access scope.

This rule is defined by the following class: `oclint-rules/rules/cocoa/ObjCVerifyProtectedMethodRule.cpp`

Example:

```
@interface A : NSObject
- (void)foo __attribute__((annotate("oclint:enforce[protected method]")));
@end

@interface B : NSObject
@property (strong, nonatomic) A* a;
@end

@implementation B
- (void)bar {
    [self.a foo]; // calling protected method foo from outside A and its subclasses
}
@end
```

6.2.5 MissingAbstractMethodImplementation

Since: 0.8

Name: missing abstract method implementation

Due to the Objective-C language tries to postpone the decision makings to the runtime as much as possible, an abstract method is okay to be declared but without implementations. This rule tries to verify the subclass implement the correct abstract method.

This rule is defined by the following class: `oclint-rules/rules/cocoa/ObjCVerifySubclassMustImplementRule.cpp`

Example:

```
@interface Parent

- (void)anAbstractMethod __attribute__((annotate("oclint:enforce[abstract method]")));

@end

@interface Child : Parent
@end

@implementation Child

/*
// Child, as a subclass of Parent, must implement anAbstractMethod
- (void)anAbstractMethod {}
*/

@end
```

6.3 Convention

6.3.1 AvoidBranchingStatementAsLastInLoop

Since: 0.7

Name: avoid branching statement as last in loop

Having branching statement as the last statement inside a loop is very confusing, and could largely be forgetting of something and turning into a bug.

This rule is defined by the following class: `oclint-rules/rules/convention/AvoidBranchingStatementAsLastInLoopRule.cpp`

Example:

```
void example()
{
    for (int i = 0; i < 10; i++)
    {
        if (foo(i))
        {
            continue;
        }
        break;        // this break is confusing
    }
}
```

6.3.2 ProblematicBaseClassDestructor

Since: 0.10.2

Name: base class destructor should be virtual or protected

Make base class destructor public and virtual, or protected and nonvirtual

This rule is defined by the following class: `oclint-rules/rules/convention/BaseClassDestructorShouldBeVirtualOrProtectedRule.cpp`

Example:

```
class Base
{
public:
    ~Base(); // this should be either protected or virtual
}
class C : public Base
{
    virtual ~C();
}
```

References:

Sutter & Alexandrescu (November 2004). “C++ Coding Standards: 101 Rules, Guidelines, and Best Practices”. Addison-Wesley Professional

6.3.3 UnnecessaryDefaultStatement

Since: 0.8

Name: unnecessary default statement in covered switch statement

When a switch statement covers all possible cases, a default label is not needed and should be removed. If the switch is not fully covered, the `SwitchStatementsShouldHaveDefault` rule will report.

This rule is defined by the following class: `oclint-rules/rules/convention/CoveredSwitchStatementsDontNeedDefaultRule.cpp`

Example:

```
typedef enum {
    value1 = 0,
    value2 = 1
} eValues;

void aMethod(eValues a)
{
    switch(a)
    {
        case value1:
            break;
        case value2:
            break;
        default:           // this break is obsolete because all
                           // values of variable a are already covered.
            break;
    }
}
```

6.3.4 MisplacedDefaultLabel

Since: 0.6

Name: ill-placed default label in switch statement

It is very confusing when default label is not the last label in a switch statement.

This rule is defined by the following class: `oclint-rules/rules/convention/DefaultLabelNotLastInSwitchStatementRule.cpp`

Example:

```
void example(int a)
{
    switch (a) {
        case 1:
            break;
        default: // the default case should be last
            break;
        case 2:
            break;
    }
}
```

6.3.5 DestructorOfVirtualClass

Since: 0.8

Name: destructor of virtual class

This rule enforces the destructor of a virtual class must be virtual.

This rule is defined by the following class: `oclint-rules/rules/convention/DestructorOfVirtualClassRule.cpp`

Example:

```
class Base { // class Base should have a virtual destructor ~Base()
public: virtual void f();
};
class Child : public Base {
public: ~Child(); // destructor ~Child() should be virtual
};
```

6.3.6 InvertedLogic

Since: 0.4

Name: inverted logic

An inverted logic is hard to understand.

This rule is defined by the following class: `oclint-rules/rules/convention/InvertedLogicRule.cpp`

Example:

```
int example(int a)
{
    int i;
    if (a != 0)           // if (a == 0)
    {                     // {
        i = 1;           // i = 0;
    }                     // }
    else                  // else
    {                     // {
```

(continues on next page)

(continued from previous page)

```

        i = 0;                //      i = 1;
    }                        // }

    return !i ? -1 : 1;        // return i ? 1 : -1;
}

```

6.3.7 MissingBreakInSwitchStatement

Since: 0.6

Name: missing break in switch statement

A switch statement without a break statement has a very large chance to contribute a bug.

This rule is defined by the following class: `oclint-rules/rules/convention/MissingBreakInSwitchStatementRule.cpp`

Example:

```

void example(int a)
{
    switch (a) {
        case 1:
            break;
        case 2:
            // do something
        default:
            break;
    }
}

```

6.3.8 NonCaseLabelInSwitchStatement

Since: 0.6

Name: non case label in switch statement

It is very confusing when label becomes part of the switch statement.

This rule is defined by the following class: `oclint-rules/rules/convention/NonCaseLabelInSwitchStatementRule.cpp`

Example:

```

void example(int a)
{
    switch (a) {
        case 1:
            break;
        labell:    // label in a switch statement in really confusing
            break;
        default:
            break;
    }
}

```

6.3.9 AssignIvarOutsideAccessors

Since: 0.8

Name: ivar assignment outside accessors or init

This rule prevents assigning an ivar outside of getters, setters, and `init` method.

This rule is defined by the following class: `oclint-rules/rules/convention/ObjCAssignIvarOutsideAccessorsRule.cpp`

Example:

```
@interface Foo : NSObject
{
    int _bar;
}
@property (assign, nonatomic) int bar;
@end
@implementation Foo
@synthesize bar = _bar;
- (void)doSomething {
    _bar = 3; // access _bar outside its getter, setter or init
}
@end
```

6.3.10 ParameterReassignment

Since: 0.6

Name: parameter reassignment

Reassigning values to parameters is very problematic in most cases.

This rule is defined by the following class: `oclint-rules/rules/convention/ParameterReassignmentRule.cpp`

Example:

```
void example(int a)
{
    if (a < 0)
    {
        a = 0; // reassign parameter a to 0
    }
}
```

6.3.11 PreferEarlyExit

Since: 0.8

Name: prefer early exits and continue

Early exits can reduce the indentation of a block of code, so that reader do not have to remember all the previous decisions, therefore, makes it easier to understand the code.

This rule is defined by the following class: `oclint-rules/rules/convention/PreferEarlyExitRule.cpp`

Example:


```

int *doSomething(int a) {
    if (!foo(a) && bar(a) && doOtherThing(a)) {
        // ... some really long code ....
    }

    return 0;
}

// is preferred as

int *doSomething(int a) {
    if (foo(a)) {
        return 0;
    }

    if (!bar(a)) {
        return 0;
    }

    if (!doOtherThing(a)) {
        return 0;
    }

    // ... some long code ....
}

```

6.3.12 MissingDefaultStatement

Since: 0.6

Name: missing default in switch statements

Switch statements should have a default statement.

This rule is defined by the following class: `oclint-rules/rules/convention/SwitchStatementsShouldHaveDefaultRule.cpp`

Example:

```

void example(int a)
{
    switch (a) {
        case 1:
            break;
        case 2:
            break;
        // should have a default
    }
}

```

6.3.13 TooFewBranchesInSwitchStatement

Since: 0.6

Name: too few branches in switch statement

To increase code readability, when a switch consists of only a few branches, it's much better to use an if statement instead.

This rule is defined by the following class: `oclint-rules/rules/convention/TooFewBranchesInSwitchStatementRule.cpp`

Example:

```
void example(int a)
{
    switch (a) {
        case 1:
            break;
        default:
            break;
    } // Better to use an if statement and check if variable a equals 1.
}
```

Thresholds:

MINIMUM_CASES_IN_SWITCH The reporting threshold for count of case statements in a switch statement, default value is 3.

6.4 Design

6.4.1 AvoidDefaultArgumentsOnVirtualMethods

Since: 0.10.1

Name: avoid default arguments on virtual methods

Giving virtual functions default argument initializers tends to defeat polymorphism and introduce unnecessary complexity into a class hierarchy.

This rule is defined by the following class: `oclint-rules/rules/design/AvoidDefaultArgumentsOnVirtualMethodsRule.cpp`

Example:

```
class Foo
{
public:
    virtual ~Foo();
    virtual void a(int b = 3);
    // ...
};

class Bar : public Foo
{
public:
    void a(int b);
    // ...
};

Bar *bar = new Bar;
Foo *foo = bar;
foo->a();    // default of 3
bar->a();    // compile time error!
```

6.4.2 AvoidPrivateStaticMembers

Since: 0.10.1

Name: avoid private static members

Having static members is easier to harm encapsulation.

This rule is defined by the following class: `oclint-rules/rules/design/AvoidPrivateStaticMembersRule.cpp`

Example:

```
class Foo
{
    static int a;      // static field
};
class Bar
{
    static int b();    // static method
}
```

6.5 Empty

6.5.1 EmptyCatchStatement

Since: 0.6

Name: empty catch statement

This rule detects instances where an exception is caught, but nothing is done about it.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyCatchStatementRule.cpp`

Example:

```
void example()
{
    try
    {
        int* m= new int[1000];
    }
    catch(...)           // empty catch statement, this swallows an exception
    {
    }
}
```

6.5.2 EmptyDoWhileStatement

Since: 0.6

Name: empty do/while statement

This rule detects instances where do-while statement does nothing.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyDoWhileStatementRule.cpp`

Example:

```
void example()
{
    do
```

(continues on next page)

(continued from previous page)

```
{                                     // empty do-while statement
} while(1);
}
```

6.5.3 EmptyElseBlock

Since: 0.6

Name: empty else block

This rule detects instances where a else statement does nothing.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyElseBlockRule.cpp`

Example:

```
int example(int a)
{
    if (1)
    {
        return a + 1;
    }
    else                                     // empty else statement, can be safely removed
    {
    }
}
```

6.5.4 EmptyFinallyStatement

Since: 0.6

Name: empty finally statement

This rule detects instances where a finally statement does nothing.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyFinallyStatementRule.cpp`

Example:

```
void example()
{
    Foo *foo;
    @try
    {
        [foo bar];
    }
    @catch(NSException *e)
    {
        NSLog(@"Exception occurred: %@", [e description]);
    }
    @finally                                     // empty finally statement, probably forget to clean up?
    {
    }
}
```

6.5.5 EmptyForStatement

Since: 0.6

Name: empty for statement

This rule detects instances where a for statement does nothing.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyForStatementRule.cpp`

Example:

```
void example(NSArray *array)
{
    for (;;)                // empty for statement
    {
    }

    for (id it in array)     // empty for-each statement
    {
    }
}
```

6.5.6 EmptyIfStatement

Since: 0.2

Name: empty if statement

This rule detects instances where a condition is checked, but nothing is done about it.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyIfStatementRule.cpp`

Example:

```
void example(int a)
{
    if (a == 1)              // empty if statement
    {
    }
}
```

6.5.7 EmptySwitchStatement

Since: 0.6

Name: empty switch statement

This rule detects instances where a switch statement does nothing.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptySwitchStatementRule.cpp`

Example:

```
void example(int i)
{
    switch (i)               // empty switch statement
    {
    }
```

(continues on next page)

(continued from previous page)

```
}  
}
```

6.5.8 EmptyTryStatement

Since: 0.6

Name: empty try statement

This rule detects instances where a try statement is empty.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyTryStatementRule.cpp`

Example:

```
void example()  
{  
    try                                // but this try statement is empty  
    {  
    }  
    catch(...)  
    {  
        cout << "Exception is caught!";  
    }  
}
```

6.5.9 EmptyWhileStatement

Since: 0.6

Name: empty while statement

This rule detects instances where a while statement does nothing.

This rule is defined by the following class: `oclint-rules/rules/empty/EmptyWhileStatementRule.cpp`

Example:

```
void example(int a)  
{  
    while(a--)  
    {  
    }  
}
```

6.6 Migration

6.6.1 UseBoxedExpression

Since: 0.7

Name: use boxed expression

This rule locates the places that can be migrated to the new Objective-C literals with boxed expressions.

This rule is defined by the following class: `oclint-rules/rules/migration/ObjCBoxedExpressionsRule.cpp`

Example:

```
void aMethod()
{
    NSNumber *fortyTwo = [NSNumber numberWithInt:(43 - 1)];
    // NSNumber *fortyTwo = @(43 - 1);

    NSString *env = [NSString stringWithUTF8String:getenv("PATH")];
    // NSString *env = @(getenv("PATH"));
}
```

6.6.2 UseContainerLiteral

Since: 0.7

Name: use container literal

This rule locates the places that can be migrated to the new Objective-C literals with container literals.

This rule is defined by the following class: `oclint-rules/rules/migration/ObjCContainerLiteralsRule.cpp`

Example:

```
void aMethod()
{
    NSArray *a = [NSArray arrayWithObjects:@1, @2, @3, nil];
    // NSArray *a = @[ @1, @2, @3 ];

    NSDictionary *d = [NSDictionary dictionaryWithObjects:@[@2,@4] forKey:@[@1,@3]];
    // NSDictionary *d = @{ @1 : @2, @3 : @4 };
}
```

6.6.3 UseNumberLiteral

Since: 0.7

Name: use number literal

This rule locates the places that can be migrated to the new Objective-C literals with number literals.

This rule is defined by the following class: `oclint-rules/rules/migration/ObjCNSNumberLiteralsRule.cpp`

Example:

```
void aMethod()
{
    NSNumber *fortyTwo = [NSNumber numberWithInt:42];
    // NSNumber *fortyTwo = @42;

    NSNumber *yesBool = [NSNumber numberWithBool:YES];
    // NSNumber *yesBool = @YES;
}
```

6.6.4 UseObjectSubscripting

Since: 0.7

Name: use object subscripting

This rule locates the places that can be migrated to the new Objective-C literals with object subscripting.

This rule is defined by the following class: `oclint-rules/rules/migration/ObjCObjectSubscriptingRule.cpp`

Example:

```
void aMethod(NSArray *a, NSDictionary *d)
{
    id item = [a objectAtIndex:0];
    // id item = a[0];

    id item = [d objectForKey:@1];
    // id item = d[@1];
}
```

6.7 Naming

6.7.1 LongVariableName

Since: 0.7

Name: long variable name

Variables with long names harm readability.

This rule is defined by the following class: `oclint-rules/rules/naming/LongVariableNameRule.cpp`

Example:

```
void aMethod()
{
    int reallyReallyLongIntegerName;
}
```

Thresholds:

LONG_VARIABLE_NAME The long variable name reporting threshold, default value is 20.

6.7.2 ShortVariableName

Since: 0.7

Name: short variable name

A variable with a short name is hard to understand what it stands for. Variable with name, but the name has number of characters less than the threshold will be emitted.

This rule is defined by the following class: `oclint-rules/rules/naming/ShortVariableNameRule.cpp`

Example:


```
void aMethod(int i) // i is short
{
    int ii;         // ii is short
}
```

Thresholds:

SHORT_VARIABLE_NAME The short variable name reporting threshold, default value is 3.

6.8 Redundant

6.8.1 RedundantConditionalOperator

Since: 0.6

Name: redundant conditional operator

This rule detects three types of redundant conditional operators:

1. true expression and false expression are returning true/false or false/true respectively;
2. true expression and false expression are the same constant;
3. true expression and false expression are the same variable expression.

They are usually introduced by mistake, and should be simplified.

This rule is defined by the following class: `oclint-rules/rules/redundant/RedundantConditionalOperatorRule.cpp`

Example:

```
void example(int a, int b, int c)
{
    bool b1 = a > b ? true : false;    // true/false: bool b1 = a > b;
    bool b2 = a > b ? false : true;    // false/true: bool b2 = !(a > b);
    int i1 = a > b ? 1 : 1;             // same constant: int i1 = 1;
    float f1 = a > b ? 1.0 : 1.00;     // equally constant: float f1 = 1.0;
    int i2 = a > b ? c : c;             // same variable: int i2 = c;
}
```

6.8.2 RedundantIfStatement

Since: 0.4

Name: redundant if statement

This rule detects unnecessary if statements.

This rule is defined by the following class: `oclint-rules/rules/redundant/RedundantIfStatementRule.cpp`

Example:

```
bool example(int a, int b)
{
    if (a == b) // this if statement is redundant
    {
        return true;
    }
}
```

(continues on next page)

(continued from previous page)

```
    else
    {
        return false;
    }
    // the entire method can be simplified to return a == b;
}
```

6.8.3 RedundantLocalVariable

Since: 0.4

Name: redundant local variable

This rule detects cases where a variable declaration is immediately followed by a return of that variable.

This rule is defined by the following class: `oclint-rules/rules/redundant/RedundantLocalVariableRule.cpp`

Example:

```
int example(int a)
{
    int b = a * 2;
    return b;    // variable b is returned immediately after its declaration,
                // can be simplified to return a * 2;
}
```

6.8.4 RedundantNilCheck

Since: 0.7

Name: redundant nil check

C/C++-style null check in Objective-C like `foo != nil && [foo bar]` is redundant, since sending a message to a nil object in this case simply returns a false-y value.

This rule is defined by the following class: `oclint-rules/rules/redundant/RedundantNilCheckRule.cpp`

Example:

```
+ (void)compare: (A *) obj1 withOther: (A *) obj2
{
    if (obj1 && [obj1 isEqualTo:obj2]) // if ([obj1 isEqualTo:obj2]) is okay
    {
    }
}
```

6.8.5 UnnecessaryElseStatement

Since: 0.6

Name: unnecessary else statement

When an if statement block ends with a return statement, or all branches in the if statement block end with return statements, then the else statement is unnecessary. The code in the else statement can be run without being in the block.

This rule is defined by the following class: `oclint-rules/rules/redundant/UnnecessaryElseStatementRule.cpp`

Example:

```

bool example(int a)
{
    if (a == 1)                // if (a == 1)
    {                          // {
        cout << "a is 1.";    // cout << "a is 1.";
        return true;          // return true;
    }                          // }
    else                       //
    {                          //
        cout << "a is not 1." // cout << "a is not 1."
    }                          //
}

```

6.8.6 UnnecessaryNullCheckForDealloc

Since: 0.8

Name: unnecessary null check for dealloc

`char* p = 0; delete p;` is valid. This rule locates unnecessary `if (p)` checks.

This rule is defined by the following class: `oclint-rules/rules/redundant/UnnecessaryNullCheckForCXXDeallocRule.cpp`

Example:

```

void m(char* c) {
    if (c != nullptr) { // and be simplified to delete c;
        delete c;
    }
}

```

6.8.7 UselessParentheses

Since: 0.6

Name: useless parentheses

This rule detects useless parentheses.

This rule is defined by the following class: `oclint-rules/rules/redundant/UselessParenthesesRule.cpp`

Example:

```

int example(int a)
{
    int y = (a + 1);    // int y = a + 1;
    if ((y > 0))         // if (y > 0)
    {
        return a;
    }
    return (0);          // return 0;
}

```

6.9 Size

6.9.1 HighCyclomaticComplexity

Since: 0.4

Name: high cyclomatic complexity

Cyclomatic complexity is determined by the number of linearly independent paths through a program's source code. In other words, cyclomatic complexity of a method is measured by the number of decision points, like `if`, `while`, and `for` statements, plus one for the method entry.

The experiments McCabe, the author of cyclomatic complexity, conclude that methods in the 3 to 7 complexity range are quite well structured. He also suggest the cyclomatic complexity of 10 is a reasonable upper limit.

This rule is defined by the following class: `oclint-rules/rules/size/CyclomaticComplexityRule.cpp`

Example:

```
void example(int a, int b, int c) // 1
{
    if (a == b) // 2
    {
        if (b == c) // 3
        {
        }
        else if (a == c) // 3
        {
        }
        else
        {
        }
    }
    for (int i = 0; i < c; i++) // 4
    {
    }
    switch(c)
    {
        case 1: // 5
            break;
        case 2: // 6
            break;
        default: // 7
            break;
    }
}
```

Thresholds:

CYCLOMATIC_COMPLEXITY The cyclomatic complexity reporting threshold, default value is 10.

Suppress:

```
__attribute__((annotate("oclint:suppress[high cyclomatic complexity]")))
```

References:

McCabe (December 1976). "A Complexity Measure". *IEEE Transactions on Software Engineering*: 308–320

6.9.2 LongClass

Since: 0.6

Name: long class

Long class generally indicates that this class tries to do many things. Each class should do one thing and that one thing well.

This rule is defined by the following class: `oclint-rules/rules/size/LongClassRule.cpp`

Example:

```
class Foo
{
    void bar()
    {
        // 1001 lines of code
    }
}
```

Thresholds:

LONG_CLASS The class size reporting threshold, default value is 1000.

6.9.3 LongLine

Since: 0.6

Name: long line

When the number of characters for one line of code is very high, it largely harms the readability. Break long lines of code into multiple lines.

This rule is defined by the following class: `oclint-rules/rules/size/LongLineRule.cpp`

Example:

```
void example()
{
    int a012345678901234567890123456789...
    ↪ 123456789012345678901234567890123456789;
}
```

Thresholds:

LONG_LINE The long line reporting threshold, default value is 100.

6.9.4 LongMethod

Since: 0.4

Name: long method

Long method generally indicates that this method tries to do many things. Each method should do one thing and that one thing well.

This rule is defined by the following class: `oclint-rules/rules/size/LongMethodRule.cpp`

Example:

```
void example()
{
    cout << "hello world";
    cout << "hello world";
    // repeat 48 times
}
```

Thresholds:

LONG_METHOD The long method reporting threshold, default value is 50.

6.9.5 HighNcssMethod

Since: 0.6

Name: high ncss method

This rule counts number of lines for a method by counting Non Commenting Source Statements (NCSS). NCSS only takes actual statements into consideration, in other words, ignores empty statements, empty blocks, closing brackets or semicolons after closing brackets. Meanwhile, a statement that is broken into multiple lines contribute only one count.

This rule is defined by the following class: [oclint-rules/rules/size/NcssMethodCountRule.cpp](#)

Example:

```
void example()           // 1
{
    if (1)                // 2
    {
    }
    else                  // 3
    {
    }
}
```

Thresholds:

NCSS_METHOD The high NCSS method reporting threshold, default value is 30.

Suppress:

```
__attribute__((annotate("oclint:suppress[high ncss method]")))
```

6.9.6 DeepNestedBlock

Since: 0.6

Name: deep nested block

This rule indicates blocks nested more deeply than the upper limit.

This rule is defined by the following class: [oclint-rules/rules/size/NestedBlockDepthRule.cpp](#)

Example:

```

if (1)
{
    // 1
    {
        // 2
        {
            // 3
        }
    }
}

```

Thresholds:

NESTED_BLOCK_DEPTH The depth of a block or compound statement reporting threshold, default value is 5.

6.9.7 HighNPathComplexity

Since: 0.4

Name: high npath complexity

NPath complexity is determined by the number of execution paths through that method. Compared to cyclomatic complexity, NPath complexity has two outstanding characteristics: first, it distinguishes between different kinds of control flow structures; second, it takes the various type of acyclic paths in a flow graph into consideration.

Based on studies done by the original author in AT&T Bell Lab, an NPath threshold value of 200 has been established for a method.

This rule is defined by the following class: `oclint-rules/rules/size/NPathComplexityRule.cpp`

Example:

```

void example()
{
    // complicated code that is hard to understand
}

```

Thresholds:

NPATH_COMPLEXITY The NPath complexity reporting threshold, default value is 200.

Suppress:

```
__attribute__((annotate("oclint:suppress[high npath complexity]")))
```

References:

Brian A. Nejme (1988). "NPATh: a measure of execution path complexity and its applications". *Communications of the ACM* 31 (2) p. 188-200

6.9.8 TooManyFields

Since: 0.7

Name: too many fields

A class with too many fields indicates it does too many things and lacks proper abstraction. It can be redesigned to have fewer fields.

This rule is defined by the following class: `oclint-rules/rules/size/TooManyFieldsRule.cpp`

Example:

```
class c
{
    int a, b;
    int c;
    // ...
    int l;
    int m, n;
    // ...
    int x, y, z;

    void m() {}
};
```

Thresholds:

TOO_MANY_FIELDS The reporting threshold for too many fields, default value is 20.

6.9.9 TooManyMethods

Since: 0.7

Name: too many methods

A class with too many methods indicates it does too many things and is hard to read and understand. It usually contains complicated code, and should be refactored.

This rule is defined by the following class: [oclint-rules/rules/size/TooManyMethodsRule.cpp](#)

Example:

```
class c
{
    int a();
    int b();
    int c();
    // ...
    int l();
    int m();
    int n();
    // ...
    int x();
    int y();
    int z();
    int aa();
    int ab();
    int ac();
    int ad();
    int ae();
};
```

Thresholds:

TOO_MANY_METHODS The reporting threshold for too many methods, default value is 30.

6.9.10 TooManyParameters

Since: 0.7

Name: too many parameters

Methods with too many parameters are hard to understand and maintain, and are thirsty for refactorings, like [Replace Parameter With method](#), [Introduce Parameter Object](#), or [Preserve Whole Object](#).

This rule is defined by the following class: `oclint-rules/rules/size/TooManyParametersRule.cpp`

Example:

```
void example(int a, int b, int c, int d, int e, int f,
            int g, int h, int i, int j, int k, int l)
{
}
```

Thresholds:

TOO_MANY_PARAMETERS The reporting threshold for too many parameters, default value is 10.

References:

Fowler, Martin (1999). *Refactoring: Improving the design of existing code*. Addison Wesley.

6.10 Unused

6.10.1 UnusedLocalVariable

Since: 0.4

Name: unused local variable

This rule detects local variables that are declared, but not used.

This rule is defined by the following class: `oclint-rules/rules/unused/UnusedLocalVariableRule.cpp`

Example:

```
int example(int a)
{
    int i;           // variable i is declared, but not used
    return 0;
}
```

Suppress:

```
__attribute__((annotate("oclint:suppress[unused local variable]")))
```

6.10.2 UnusedMethodParameter

Since: 0.4

Name: unused method parameter

This rule detects parameters that are not used in the method.

This rule is defined by the following class: `oclint-rules/rules/unused/UnusedMethodParameterRule.cpp`

Example:

```
int example(int a) // parameter a is not used
{
    return 0;
}
```

Suppress:

```
__attribute__((annotate("oclint:suppress[unused method parameter]")))
```

7.1 Core Module

Core module is the engine of OCLint. In general, it dispatches the tasks to other modules in order, drives the work flow of the entire analysis process, and generates outputs.

When OCLint is invoked, it reads the source code files and parses them into abstract syntax tree (AST), it also loads rules and chosen reporter.

Rules then analyze deeply into each AST representation of the source code, and find smelly AST nodes that match the predefined patterns.

During the inspection, certain metrics are calculated to have a better understanding of the source code. In addition, some rules ask for threshold configurations in order to decide whether to emit violations.

All violations are collected into a set, and are eventually sent to the reporter. Selected reporter will then convert violation set and related statistic summarization into certain type of report for output.

Following is a brief description of each component in the core module:

CommandLineOptions Extends command line arguments on top of LibTooling and defines new options for OCLint.

Processor Takes over `ASTContext`, initializes `ViolationSet` and `RuleCarrier`, and then applies all loaded rules for analysis, finally, pushes the `ViolationSet` into `Results`.

Reporter Defines the interface of all reporters.

Results Is essentially a collection of `ViolationSet`. Plus some methods to find statistic results inside all violations.

RuleBase Defines the base interface of all rules.

RuleCarrier As its name implies, it carries the AST of the source code and a violation set. When it is initialized in `Processor`, it takes an immutable AST context and an empty violation set. Then, it is examined through all rules. Each rule traverses the AST context and focus on a specific aspect. When it finds a matched pattern, a violation will be pushed to violation set. Eventually, a *contaminated* `RuleCarrier` will return back with all violations for diagnostic.

RuleConfiguration Statically stores custom thresholds for rules.

RuleSet When selected rules are loaded, they are collected into this `RuleSet`.

Version Contains the current version identifier, can be used in places, like reporters, where version is a differentiator.

Violation Is a container of all details about a violation, like the violated rule, path of the source code, start line and column number, end line and column number, and sometimes, a helper diagnostic message or suggestion.

ViolationSet Is a collection of violations.

7.2 Metrics Module

Metrics module is an isolated library. This module actually doesn't depend on any other OCLint modules. It means we can use this module separately in other projects that also measure the metrics of the source code.

7.2.1 Cyclomatic Complexity

Cyclomatic complexity was introduced by Thomas McCabe. It is determined based on the number of decision points in a method.

7.2.2 NPath Complexity

NPath complexity measures the number of acyclic execution paths in a method. It differs from cyclomatic complexity as it takes into account the nesting of conditional statements and multi-part boolean expressions.

7.2.3 Non Commenting Source Statements

Non commenting source statements (NCSS) counts the number of all statements excluding comments, empty statements, empty blocks, closing brackets or semicolons after closing brackets.

7.2.4 Statement Depth

Statement depth is the number of nesting levels.

7.3 Rules Module

OCLint is a rule based tools. Rules are dynamic libraries that can be easily loaded into system during runtime. It largely makes the tool very extensible. In addition, by following [Open/Close Principle](#), OCLint is very open to new rules by dynamically loading extended rules without modifying or recompiling itself,

All the rules are implemented as a subclass of `RuleBase`. They generally fall in two big categories - rules by reading the source code line by line, and the ones by recognizing patterns in the abstract syntax tree (AST).

Rules that belong to the first category can leverage the abstract class `AbstractSourceCodeReaderRule` to go through each line of the source code.

For rules that are interested in AST, we provide two detail approaches - AST visitor and AST matcher.

In AST visitor approach, by following the [Visitor Pattern](#), the entire AST is traversed recursively from the root of the tree. Each node is visited in a [depth-first preorder traversal](#). The visitor usually returns after all nodes are visited.

However, we can interrupt the traversal by intent, for example, when we are only curious about if certain patterns exist rather than how many of them, in this case, whenever that pattern is matched, we could stop the visitor to avoid wasting more resources, so that the performance can be improved.

Tree traversal is very mature and powerful, we can achieve almost everything with it. But it's not that intuitive to some extents. So, AST matcher, on the other hand, can help write lightweight code with better readability.

The way to think about AST matcher to AST is like XPath to XML. Specific patterns in AST are described in a simple, concise, and descriptive representation. Although it's implemented by AST visitor under the hook, the matcher expression makes it more friendly when people who read the code try to understand what the rule actually does. For example, in order to find an if statement, we can simple write our matcher like

```
ifStmt()
```

When we want to narrow our search criteria for a particular group of if statements, let's say, a if statement with a binary operator that compares if two variables are equal, then we can extend the above matcher to

```
ifStmt(hasCondition(binaryOperator(hasOperatorName("&&"))))
```

It can be read just like a sentence.

Using AST matchers has more restrictions than AST visitor, and it takes much longer time in analysis, this could lower the performance of the tool. So, we always have to consider the trade-off, and choose wisely.

7.4 Reporters Module

After the analysis process, for each of the violations detected, we know the detail information of the node, the rule, and sometimes the diagnostic message. Reporters will take the information, and convert them into human readable reporters.

Note: In some cases, the outputs are not directly readable for people, instead they will be rendered by other tools for better representations. For example, continuous integration systems can understand a specific type of output, and convert it into graphic interface on its dashboard.

8.1 System Requirements

OCLint can be compiled on all unix-like systems, theoretically. Latest macOS, FreeBSD, and major Linux distributions are tested and recommended.

8.1.1 LLVM System Requirements

OCLint is based on [libTooling](#) for parsing source code and generating [Abstract Syntax Tree \(AST\)](#) generation. Prior to compiling OCLint, we need to compile LLVM/Clang. Check out [LLVM System Requirements](#) for requirements. These requirements are very fundamental development tools that probably you already have them.

8.1.2 OCLint System Requirements

Some tools are emphasized here, you may want to double check and make sure the toolkit is ready when you need it.

1. [Apache Subversion](#)
2. [Git](#)
3. [CMake](#)
4. [LTP GCOV Extension](#) (lcov)
5. [OpenSSL](#) (only for analytics-enabled build)

8.1.3 Extra Notes

Notes for macOS

[Homebrew](#) is highly recommended for macOS users. When we have `homebrew` installed, we can simply get all the dependencies by

```
brew install subversion git cmake lcov openssl
```

Notes for Ubuntu

All Ubuntu users should be able to get all the dependencies by `apt-get`, for example:

```
apt-get install subversion git cmake lcov libssl-dev
```

Notes for Fedora

We should be able to get all the dependencies by `yum`, for example:

```
yum install gcc-c++ make git subversion cmake lcov openssl-devel
```

Notes for Python 2.6

`argparse` module is required, and install it by

```
sudo easy_install argparse
```

The installation of `argparse` module can be checked by

```
python -c "import argparse; print argparse"
```

8.2 Checking Out Code

This document helps get the latest source code of OCLint, its submodules and dependencies. We will also describe the preferred structure of your codebase in this document.

Note: There is a summary of necessary terminal commands at the bottom of this document, here is the free ride for skipping all the detail and *going directly to there*.

8.2.1 OCLint GitHub Repositories

All source code of OCLint and its submodules are hosted at [GitHub](#).

oclint/oclint

This repository hosts the OCLint source code, which consists of core, metrics, rules, and reporters modules. Check out the code by

```
git clone https://github.com/oclint/oclint.git
```

It will create a `oclint` folder in current directory, let's give it an alias name `OCLINT_HOME`.

oclint/oclint-json-compilation-database

This repository contains the Python code for `oclint-json-compilation-database`.

Go into the `OCLINT_HOME` folder, and check out this repository in there:

```
git clone https://github.com/oclint/oclint-json-compilation-database.git
```

oclint/oclint-xcodebuild

Code for `oclint-xcodebuild` has been pushed to this repository. Go to the `OCLINT_HOME` folder, and check out the code:

```
git clone https://github.com/oclint/oclint-xcodebuild.git
```

Note: `oclint/oclint` is pretty much everything you need if you are interested in OCLint development. The other two helper programs support OCLint to ease developers in certain ways. They are standalone programs themselves. Check out them when you want to improve them or you are willing to use the development version OCLint in your environment. Also, in order to run [dogfooding](#), you still need to checkout `oclint/oclint-json-compilation-database`.

8.2.2 LLVM/Clang Codebase

LLVM/Clang has its own code structure, and the detail information can be found at Clang's [Get Started](#) page. We also provide a script that checks out all LLVM/Clang repositories into correct location.

```
cd oclint-scripts
./clang checkout
cd ..
```

In addition, `clang` script does more than that:

- First, `./clang update` can update the existing LLVM/Clang checkout.
- Second, you can check out a branch codebase other than the trunk codebase by `./clang checkout -branch <branch_name>`

8.2.3 countly-cpp

We use `countly-cpp` for sending analytics collections, so you only need this if you build OCLint with analytics enabled.

```
cd oclint-scripts
./countly checkout
cd ..
```

8.2.4 googletest/googlemock

Google C++ Testing and Mocking Frameworks are used for testing OCLint. OCLint follows [Test Driven Development](#) (TDD), so checkout them before we work on this codebase and want to make sure the modifications do not break the other pieces of code.

We also provide a script `googleTest`:

- Check out the code simply by `./googleTest checkout`
- Update the codebase by `./googleTest update`

8.2.5 Summary

Sum up, to check out all OCLint modules and dependencies, we could execute the following commands:

```
git clone https://github.com/oclint/oclint.git
cd oclint
git clone https://github.com/oclint/oclint-json-compilation-database.git
git clone https://github.com/oclint/oclint-xcodebuild.git
cd oclint-scripts
./clang checkout
./countly checkout
./googleTest checkout
cd .. # back to the root folder of OCLint codebase
```

To update the entire codebase, we can do:

```
cd oclint # start from OCLint root directory
git pull origin master
cd oclint-json-compilation-database
git pull origin master
cd ../oclint-xcodebuild
git pull origin master
cd ../oclint-scripts
./clang update
./googleTest update
cd .. # back to OCLint root directory
```

So now, we OCLint directory might be like this:

```
oclint
|-README
|-build
|-countly
|-googletest
|-llvm
|-oclint-core
|---include
|---lib
|---test
|-oclint-driver
|---include
|---lib
|-oclint-json-compilation-database
|-oclint-metrics
|---include
|---lib
|---test
|-oclint-rules
|---include
|---lib
|---rules
```

(continues on next page)

(continued from previous page)

```
|---template
|---test
|-oclint-reporters
|---reporters
|---template
|---test
|-oclint-scripts
|-oclint-xcodebuild
```

8.3 Compiling and Testing

We will go through some steps to build dependencies, compile and test OCLint in this document. In general, if you have followed the codebase structure as described in [this document](#), we have scripts to gather all required steps and run them in order to facilitate the entire process.

Warning: if you need a release version OCLint binary for production use, please read [Building OCLint](#) instead.

Note: This document presumes your current working directory is `oclint/oclint-scripts`.

8.3.1 Building LLVM/Clang

We could build LLVM/Clang in debug mode with assertions by

```
./clang build
```

Debug mode binaries contain additional data to aid debugging, but lowers program performances. It's recommended to build Clang in debug mode when we work on the analysis engine, metrics, and rules. These debug information may help a lot in some circumstances.

On the other side, if our work is related to violations, reporters, and surrounding submodules, we can still use debug mode, and we can also choose release mode that enables optimizations for better performance. To build LLVM/Clang in release mode, append `release` to the script as

```
./clang build -release
```

It takes a while to build LLVM/Clang (probably much longer than a cup of coffee time). By default, this script builds the code by simultaneously using all the CPU resources. But this can be changed by explicitly specifying number of concurrent processes for the compilation, like

```
./clang build -j <num>
```

The LLVM/Clang build can be found at `oclint/build/llvm` directory, and its installation is located at `oclint/build/llvm-install` folder.

8.3.2 Building countly-cpp

We provide a script to build countly-cpp without hassle. Again, you only need this if you build OCLint with analytics enabled.

```
./countly build
```

8.3.3 Building googletest/googlemock

Building Google C++ Testing and Mocking Frameworks is easy:

```
./googleTest build
```

8.3.4 Compiling and Testing OCLint

OCLint uses CMake as build system. We can find CMakeLists.txt in each module and its include sub-folders.

`test` script would compile and test OCLint core module, metrics module, rules module, and reporters module. With module name given, the script tests only for the particular module, otherwise, it tries to test all modules. Core module and metrics module can be compiled independently. However, rules module depends on both core and metrics modules, and reporters module depends on core module. When `test` script works with all modules, the required sequence is automatically enforced.

`test` script has a `-clean` option to remove old build intermediate files.

`test` script shows the CMake configuration process, compilation progress, and test results in order. It generates code coverage report in the end. When something goes wrong, scripts stop immediately. The possible reason of a failed build could be:

- CMake configuration failed
- Build failed
- Test failed
- Code coverage failed

Testing All Modules

Test every modules:

```
./test
```

Test all modules with clean build:

```
./test -clean
```

Testing Core Module

Test core module:

```
./test core
```

Test core module with clean build:

```
./test core -clean
```

Testing Metrics Module

Test metrics module:

```
./test metrics
```

Test metrics module with clean build:

```
./test metrics -clean
```

Testing Rules Module

Test rules module:

```
./test rules
```

Test rules module with clean build:

```
./test rules -clean
```

Testing Reporters Module

Test reporters module:

```
./test reporters
```

Test reporters module with clean build:

```
./test reporters -clean
```

Reviewing Test Results

We could always go back and review our test results (unless we have cleaned test directory with `-clean` option or delete that folder manually). There is an easy way to do it with `-show` option to the `test` script.

By default, it shows the test results for all modules. We can also explicitly specify the module name as an option to it. For example, show test result for all modules:

```
./test -show
```

Show test results for core module:

```
./test core -show
```

Show test results for metrics module:

```
./test metrics -show
```

Show test results for rules module:

```
./test rules -show
```

Show test results for reporters module:

```
./test reporters -show
```

8.4 Writing Custom Rules

It's cool to add capability to OCLint by writing your own rules, instead of waiting for us get around to implementing them. You are more welcome to share your rules with the entire community.

Note: It might be easier to get started with looking at existing rules.

Rules must implement `RuleBase` class or its derived abstract classes. Different rules are specialized in different abstract levels, for example, some rules may have to dig very deep into the control flow of the code, and on the contrary, some rules only detect the defects by reading the string of the source code. [Rules module internals](#) can help pick up the right category when writing rules. Here, we skip that discussion, and directly jump to these different types of rules.

8.4.1 Generic Rules

Writing a generic rule, we need to implement `RuleBase` interface.

We name the new rule and set the priority to it. Name will be shown in the report along with the violations. Priority stands for how severe when something breaks the rule. Priorities are defined in number, the less number has a higher priority, meaning, people needs to pay more attention to them.

`RuleBase` only provides a `RuleCarrier`, which carries the context of the abstract syntax tree (`ASTContext`) and a violation set. The `ASTContext` already have all syntactic information we need for analysis, and we need to figure out the way ourselves. We write our logic in `apply` method.

When we find the nodes we need in the `ASTContext`, throw those nodes into the violation set inside `RuleCarrier`.

For all rules, we use static constructor to register the rules. A static attribute is a variable that has been allocated statically, the lifetime of static variable extends across the entire execution of the program. So, each rule implementation has a static member `rules`, and we need to make sure our rule can be collected by it when being loaded.

Writing generic rules is very flexible, we can do everything we want from the `ASTContext` we have. However, since the most of our rules can reuse certain logic to have the work done better, so we recommend the new rules inherent from the abstract classes described below instead of using `RuleBase` directly. Don't worry of losing the flexibility, all abstract classes are subclasses of `RuleBase`, so when we need this type of flexibility, we can still have it.

In addition, certain methods are still pure in these abstract classes, like `name` and `priority` methods. Plus the static `RuleSet` constructor, we still need to implement them in all rules. [Thinking about the paragraph to make it go smoother to the next paragraph.] Now, let's look at some abstract rule classes.

8.4.2 Source Code Reader Rules

`AbstractSourceCodeReaderRule` provides a `eachLine` method. We can have the text of each line and the current line number. We can then work around with the text. For example, we can calculate the length of the text, we can understand if it is a comment, we can find out if there is a mix use of spaces and tabs, and so on.

8.4.3 AST Visitor Rules

The majority of the existing rules inherit `AbstractASTVisitorRule` class. It follows the [visitor pattern](#). In our rules, we only write methods for the type of nodes that we are interested. For example, when we want to inspect all if statement, we should write

```
bool VisitIfStmt (IfStmt *ifStmt)
{
    // do stuff with this if statement
    return true;
}
```

Similarly, when we analyze an Objective-C method, we can visit that node by

```
bool VisitObjCMethodDecl (ObjCMethodDecl *decl)
{
    // analyze decl
    return false;
}
```

The return boolean value of these `visit` methods are used to control the traversal. AST visitor will continue its sub nodes or sibling nodes when visiting the current node return a true, vice versa, it stops when current `visit` method returns a false.

See also:

We are using the abstract syntax tree generated by Clang, so understanding the API of Clang AST is very helpful when writing rules. There are some useful links that we have assembled together in [Related Clang Knowledge Base](#) page.

We also have a `setUp` method and a `tearDown` method in case we need to prepare certain stuff ready before the analysis and wrap up results afterwards. `setUp` method is guaranteed to be called before visiting any of the AST nodes, and after the AST analysis is done, `tearDown` is called for sure.

8.4.4 AST Matcher Rules

If possible, we are more willing to write AST matcher rules (unless performance is a big issue), in order to achieve the great readability that comes along with the AST matcher. We always prefer simple and concise.

All AST matcher rules inherit from `AbstractASTMatcherRule` class.

We need to add all matchers in `setUpMatcher` method. With each matcher, we bind a unique (within the scope of current rule) name for that in matcher.

Then whenever a match is found, `callback` method is called with that AST node as a parameter. So we can get that node with the name we have binded in previous step. We then add this node with other information into violation set.

See also:

Again, `LibASTMatcher` is provided by Clang, and we would like to suggest you by reading some [related Clang knowledge](#) to have a better understanding.

8.4.5 Creating Rules with Scaffolding

Rules scaffolding is a quick way to create custom rules. When we want to create our custom rules and build them along with the OCLint building pipeline, scaffolding is the tool for the job.

We can tell the category, rule type, name, and priority to the scaffold script, or we can leave them with default settings.

Read on [rule scaffolding](#) document for details.

8.4.6 Build it and Make it Live

After coding for our new rule, now we have our new rule ready. We need to compile it into a dynamic library and link against `LLVMSupport`, `clangASTMatchers`, `OCLintMetric`, `OCLintUtil`, and `OCLintCore` libraries. We also have a CMake macro `build_dynamic_rule` to ease this process.

We copy the new dynamic library into `$(/path/to/bin/oclint)/../lib/oclint/rules`, and it will be loaded together with all other rules in this folder.

8.4.7 Unit Testing

We have a series of convenient methods for rules' unit testing. They are `testRuleOnCode` method for regular C code, `testRuleOnCXXCode` method for C++ code, and `testRuleOnObjCCode` method to test Objective-C code. By giving the code we want to apply the rule on and our expectation result, this method will parse the code and run only current rule, and compare the expectation. It fails the test when the rule doesn't meet expecting behaviors. A quick sample usage is like this

```
TEST(BitwiseOperatorInConditionalRuleTest, BitwiseOrInWhile)
{
    testRuleOnCode(new BitwiseOperatorInConditionalRule(), "void m() { while (1 | 0)
→{;} }", 0, 1, 19, 1, 23);
    // testRuleOnCode(
    //     new RuleToBeTested(),
    //     "source code",
    //     violationIndex,
    //     expectStartLine,
    //     expectStartColumn,
    //     expectEndLine,
    //     expectEndColumn,
    //     optionalExpectMessage);
    // When we expect the code has no violation, simple write
    // testRuleOnCode(new RuleToBeTested(), "source code");
}
```

8.5 Writing Custom Reporters

Currently we have multiple types of reporters, like plain text and HTML. To enable extended capabilities, custom reporters are easily implemented for your own supports.

We have a `oclint-reporters` module, and all reporters are recommended to add to this module.

New reporters need to inherit `Reporter` interface. We will implement two methods that are required by the interface.

First of all, give the new reporter an identifier in `name` method. Then give the same name with `-report-type` option to `oclint` main program in order to use this new reporter later.

Then implement logic in `report` method. We have a `ostream` as a output stream on hand. In addition, all the information the new reporter needs is in the `Results` class.

Lastly, there is one small extra effort that is not defined in the interface, but is required when OCLint tries to load this reporter. Please copy and paste the code below, and replace `YourNewReporter` with your class name.

```
extern "C" Reporter* create()
{
    return new YourNewReporter();
}
```


That's it. Now compile your new reporter and link it against `OCLintCore` library into a new dynamic library. We have a CMake macro `BUILD_DYNAMIC_REPORTER` to make this part easier.

See also:

[Reporter scaffolding](#) is the tool for accelerating this process, it helps create the reporter source code from template and corresponding build configurations. As developers, we only need to think about the logic of rendering results.

Put the generated dynamic library into `$(/path/to/bin/oclint)/../lib/oclint/reporters` along with other reporter libraries. Done!

8.6 Scaffolding

Rules and reporters classes are designed by contract, meaning they need to inherit their base classes, and implement pure virtual methods, and their source code follow certain structures. So we could scaffold new rules and reporters by copying their templates and replacing the placeholders. In addition, the scaffolding scripts can ease your work by adding the new rule and reporter to the OCLint build pipeline.

This document provides information on how to create rules and reporters with scaffold scripts.

8.6.1 Creating Rules with Scaffolding

Creating a custom rule can be done with `scaffoldRule` script under `oclint-scripts` folder.

We could get a list of its options by typing `./scaffoldRule -h`:

```
usage: scaffoldRule [-h] [-t {Generic,SourceCodeReader,ASTVisitor,ASTMatcher}]
                  [-c RULE_CATEGORY] [-n RULE_NAME] [-p {1,2,3}] [--test]
                  [--no-test]
                  class_name

positional arguments:
  class_name           class name of the rule

optional arguments:
  -h, --help           show this help message and exit
  -t {Generic,SourceCodeReader,ASTVisitor,ASTMatcher}, --type {Generic,
  ↪SourceCodeReader,ASTVisitor,ASTMatcher}
  -c RULE_CATEGORY, --category RULE_CATEGORY
  -n RULE_NAME, --name RULE_NAME
  -p {1,2,3}, --priority {1,2,3}
  --test              Generate a test for the new rule (default)
  --no-test           Do not generate a test for the new rule
```

From where, we could specify the class name, along with the name, type, category and priority of the rule.

Class name is the only required argument. Without explicitly given a rule name, it could be generated according to the class name. The default values of type, category and priority are `Generic`, `custom` and `3` respectively.

See also:

Learn how to choose the proper rule interface from [Rule Module Internals](#) document.

For example, if we want to create a controversial rule that extracts all switch statements with abstract syntax tree (AST) matcher, we can enter this command in the terminal:

```
./scaffoldRule AllSwitchStatements -c controversial -t ASTMatcher
```

Notice we would like scaffold script to populate the rule name and assign the default priority to this rule for us.

The scaffold script will create `AllSwitchStatementsRule.cpp` file, and store it in `controversial` folder under `oclint-rules/rules`. The rule will be generated like the following:

```
#include "oclint/AbstractASTMatcherRule.h"
#include "oclint/RuleSet.h"

class AllSwitchStatementsRule : public AbstractASTMatcherRule
{
private:
    static RuleSet rules;

public:
    virtual const string name() const
    {
        return "all switch statements";
    }

    virtual int priority() const
    {
        return 3;
    }

    virtual const string category() const override
    {
        return "controversial";
    }

    virtual void callback(const MatchFinder::MatchResult &result)
    {
    }

    virtual void setUpMatcher()
    {
    }
};

RuleSet AllSwitchStatementsRule::rules(new AllSwitchStatementsRule());
```

In addition, related `CMakeLists.txt` files will be edited to ensure the new rule will be built along with other existing rules.

A unit test file for this rule is scaffolded along with this process for testing purposes.

Now, the scaffolding is finished, we can refer to [Writing Custom Rules](#) document to fill in the logic for the rule.

8.6.2 Creating Reporters with Scaffolding

Scaffolding a reporter is very similar to the rule, but much easier, since it only requires the reporter's class name with an optional argument for specifying the reporter's name. We could also get these options by typing `./scaffoldReporter -h`:

```
usage: scaffoldReporter [-h] [-n REPORTER_NAME] [--tests] [--no-tests]
                        class_name
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  class_name          class name of the reporter

optional arguments:
  -h, --help          show this help message and exit
  -n REPORTER_NAME, --name REPORTER_NAME
  --tests             Generate a test for the new reporter (default)
  --no-tests          Do not generate a test for the new reporter
```

Let's say we want to create a new `ColorfulTextReporter`, with this script, we could do

```
./scaffoldReporter ColorfulText -n color
```

The generated `ColorfulTextReporter.cpp` will look like the following:

```
#include "oclint/Reporter.h"

class ColorfulTextReporter : public Reporter
{
public:
    virtual const string name() const
    {
        return "color";
    }

    virtual void report(Results *results, ostream &out)
    {
    }
};

extern "C" Reporter* create()
{
    return new ColorfulTextReporter();
}
```

Sequentially, the `CMakeLists.txt` file under `reporters` folder will be edited by appending the new reporter.

A unit test file for this reporter is scaffolded along with this process for testing purposes.

Now, we can refer to the [Writing Custom Reporters](#) document to print out the analysis results.

8.7 Eating Your Own Dog Food

OCLint, being a static code analysis tool that helps maintain high code quality, always applies itself against its own source code to demonstrate the quality and capabilities of itself. This process is called dogfooding. Dogfooding gives us confidence. At the same time, when we are careless and don't pay attention to certain things, dogfooding alerts us immediately.

For every software products, it's impossible to say they are bug-free or smell-free. But dogfooding is a very good practice to lower the change of being exploded to defects. Along with the growth of OCLint rule set, we always keep our code away from violating any of these rules.

8.7.1 Requirements

We always highly recommend building the local OCLint version, and then use it for dogfooding the very codebase that this binary is built from. We also need to be the latest debug version of OCLint with assertions on.

The easiest way of preparing the qualified binary is by running the scripts below. It actually invokes the `dogFooding` process automatically when the binary is ready.

```
cd oclint-scripts
./ci -reset
./ci -setup
```

8.7.2 Dogfooding

Kick off the dogfooding process by calling `dogFooding` script in `oclint-scripts` folder, like

```
cd oclint-scripts
./dogFooding
```

This will use CMake to configure the modules with `CMAKE_EXPORT_COMPILE_COMMANDS` on for generating `compile_commands.json` files. `oclint-json-compilation-database` follows after that to analyze the source code and generate dogfooding reports. It takes a while to finish the entire inspection process. When it's done, the report will be outputted on terminal.

By default, `dogFooding` script analyzes all modules. The script could work for one single module by the given module name.

Enabling Clang Static Analyzer

During the dogfooding process, it's also possible to enable Clang Static Analyzer to analyze the codebase in addition to OCLint itself.

Pass `-enable-clang-static-analyzer` option to the script, and it redirects the message to OCLint for enabling the integrated Clang Static Analyzer.

Reviewing Dogfooding Reports

By passing `-show` option to the `dogFooding` script, it prints out the existing dogfooding reports. Again, by default, it shows all modules, and can be more specific by given a module name.

Note: Please help to keep your fork of OCLint codebase free from code smells. Before submit a patch or open a pull request, please make sure to run dogfooding and fix all violations. Great thanks.

8.8 Open Projects

This page shows the the roadmap of this project. We warmly welcome all sorts of contributions. Please use [oclint-dev mailing list](#) for the best discussion channel, and use it for more ideas and suggestions.

See also:

Tasks in the pipeline, under discussion and being implemented can be found at [GitHub Issues](#) page.

8.8.1 More Rules

Rules are always welcome.

8.8.2 More Reporters

Reporters are also great welcome. We also plan to support major continuous integration systems, e.g. TeamCity and CruiseControl, by adding reporters and/or implementing plugins.

8.8.3 More Metrics

More interesting metrics can help developers measure their source code in different aspects. In addition to existing metrics, we are planning to add more metrics, like Weighted Method Count (WMC), Tight Class Cohesion (TCC), Access to Foreign Data (ATFD), Assignment-Branch-Condition (ABC), and so on.

OCLint's metrics module is actually a cohesive library, it can be reused in your project without any OCLint dependencies. So, the new introduced metrics should only depend on abstract syntax tree representation.

8.8.4 Package Manager Build

We hope to provide native support for major package managers, like homebrew for Mac and APT for Debian and its derived distributions.

8.8.5 Xcode Plugin/Add-on/Extension

We are happy with existing `oclint-xcodebuild`, but we also want to provide a graphic interface for Xcode users. The idea is either being a Xcode plugin to be able to invoke OCLint inside Xcode, or being a standalone extension to open a Xcode workspace/project, select scheme, target, and other settings, then show source code with inline analysis results.

8.8.6 Research Interests

OCLint started as a research project, so some ideas here can be explored much more with critical thinking. In addition, the tool can continue leveraging the research conclusions for higher accuracy, better performance, more reasonable software metrics, and so on.

Source Code Metrics More metrics could be explored and benefit the quality of source code.

Control Flow Graphic Engine A strong control flow graphic engine can help with a better understanding of the order that statements, expressions, and functions are evaluated or executed. For instance, goto statements force program to jump to a different statement, if statements execute a branch of statements only if certain conditions are met, while statements run the code inside a loop iteratively, etc. It helps increase the analysis accuracy and avoid false positives. We have applied control flow analysis in some existing rules, but we would like to have an engine to gain a more comprehensive understanding of the source code.

Data Flow Engine Data flow provides global information about how a procedure manipulates its data, such as the use, declaration and dependency of the data. For example, a new value could be assigned to an existing variable, and the memory address of a pointer can be reallocated. It is easier to detect the data flow in runtime. However, certain dynamic behaviors of the data can be also determined in static code analysis with data flow engine. Data flow engine is a big supplement to control flow engine.

Code Duplication Detection Duplicated code is problematic, the command text-based duplication detection could be highly improved with source code logic based duplication detection.

Refactoring Suggestions In addition to detecting code defects that break our defined rule set, we hope to provide you suggestions of how to improve your code by refactoring. We hope to do it smartly and intelligently that the suggestions will be given after fully analyzing the context of the rot code.

Design Patterns Recognition It's helpful to know the design patterns we have applied in our codebase.

Type Inference We know we can sometimes cheat to let compilers happy with certain things. This is a very dangerous practice. But, on the other hand, sometimes, we also want to take the advantages of the dynamic of programming languages, like void pointer in C and C++, and `id` in Objective-C, even though they are static typed languages. In fact, we believe type inference, the technique of automatically deduce, either partially or fully, the type even at compile time. However, this largely increase the false positive in static code analysis. Luckily, as the development of programming language techniques, type inference is introduced as a technique to automatically deduce, either partially or fully, the type of an expression at compile time. Many static typed languages have type inference capabilities builtin nowadays, so that as a developer, even though it's a static typed language, but you could omit the type annotations without explicitly telling the compiler the type. In this case, we can apply the same technique in static code analysis in order to lower false positive, and improve the accuracy at the same time.

8.9 Related Clang Knowledge Base

OCLint uses Clang APIs heavily. This includes Tooling, Abstract Syntax Tree, and Matchers libraries. A certain level of prerequisite knowledge will be great helpful when developers work on OCLint, especially for people who [write own custom rules](#).

Here is a list of links that comprehensively explain how these Clang libraries work, hope these could ease the development.

8.9.1 LibTooling

- <http://clang.llvm.org/docs/LibTooling.html>
- <http://clang.llvm.org/docs/IntroductionToTheClangAST.html>
- <http://clang.llvm.org/docs/JSONCompilationDatabase.html>

8.9.2 Clang AST

- <http://clang.llvm.org/docs/IntroductionToTheClangAST.html>
- <http://clang.llvm.org/docs/RAVFrontendAction.html>

8.9.3 LibASTMatchers

- <http://clang.llvm.org/docs/LibASTMatchers.html>
- <http://clang.llvm.org/docs/LibASTMatchersTutorial.html>
- <http://clang.llvm.org/docs/LibASTMatchersReference.html>

8.10 Coding Standards

Please help us with consistent coding standards for a number of reasons:

- Reduce the maintenance cost
- Improve the readability
- Ship source code as a product, in fact, it's an art

Although there is no restricted rules to be followed in all instances, it's still highly recommended that when you implement new code or fix existing code, use the same style that is already being used in other places of the codebase.

8.10.1 Naming Convention

In general, use [camel case](#).

Class names Should be nouns and start with an upper case letter, like `EmptyIfStatementRule`.

Variable names Should be nouns, and start with a lower case letter, like `ifStmt`. Names for class attributes (fields, members) are recommended with an underscore as prefix, like `_carrier`.

Function names Should be verbs, start with a lower case letter, like `applyLoopCompoundStmt`.

8.10.2 Use Spaces Instead of Tabs

8.10.3 Treat Compiler Warnings Like Errors

8.10.4 Don't Use `else` after `return`

Having `else` or `else if` after statements that interrupt control flow, like `return`, `break`, `continue`, etc, is confusing. It warms the readability, and sometimes, it's hard to understand. We have a rule implemented about this, so [dogfooding](#) can help identify them.

8.10.5 Don't Use Inverted Logic

To make code easier to understand, code like

```
if (!foo())
{
    doSomething();
}
else
{
    doSomethingElse();
}
```

Should be changed to

```
if (foo())
{
    doSomethingElse();
}
else
```

(continues on next page)

(continued from previous page)

```
{  
    doSomething();  
}
```

8.10.6 No States in Rules

Modified BSD License

Copyright (C) 2010-2014 Longyi Qi, 2015-2018 Ryuichi Laboratories. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Bibliography

- [McCabe76] McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. doi:10.1109/TSE.1976.233837