Optimize the Content and the Scope of the Analysis with Kiuwan Local Analyzer

How to optimize the content and the scope of the analysis with Kiuwan Local Analyzer (KLA)

Select the right source code to analyze

Before you start an analysis with KLA, you have to provide a source code directory. All the files available in this directory will be analyzed. The size of the source code to be analyzed affects proportionally the time and memory used for the analysis execution.

Avoiding analyzing unneeded code is the first approach to reduce time and memory.

See our guide on Setting Source Code Filters with KLA.

As a rule of thumb, big source files are good candidates to be excluded from the analysis, for example:

- Auto-generated code;
- Library components;
- Database exports.

To identify these large files,find the *discovery.diagnosis.txt* file in the temp directory of your analysis. It will show:

- The number of files to analyze for every technology;
- Any files bigger than a preconfigured threshold (200Kb).

Execute a ruleset according to your needs

By default, the rules analysis steps executes all of the model's active rules for every file.

The default model (CQM) contains aprox 900 rules, of which 700 are active. This means that for every file, 700 rules will be executed on their source code.

Choose a model that suits your needs the most, activating only the rules that are important to you.

A large set of rules will generate defects for high-priority rules as well as for low-priority ones.

Read more in our Guide to Model Management.

Mute or deactivate a rule

Rules can be either muted or deactivated.

Muting a rule means that the rule will still be executed in the background, however the results will be hidden (e.g. in the event of many false positives)

Deactivating a rule means that the rule will not be executed at all (e.g. found defects are uninteresting or do not apply to your application). Deactivating rules speeds up the analysis process and make it more manageable.

Read more in our Guide to Model Management.

Process JSP in Java analyses

If you are analyzing Java, there's a configuration option that has a considerable impact on analysis performance and memory needs:

- *process JSP as Java servlets?*

If this option is set to true (the default value), for every JSP Kiuwan will internally generate its java servlet code and will execute the java rules to it.

This servlet code generation consumes a considerable amount of time and memory.

The advantage to generate it is a higher precision in detecting Code Security vulnerabilities spread between JSPs and Java files (mainly XSS).

If this is not your concern, you can **set this property to false** and the execution will be faster and will run with less memory needs.

Pay attention to ambiguous file extensions

Kiuwan associates source files and technologies through file extensions and there are some extensions that are commonly associated to more than one technology. Some examples:

- .sql matches PL_SQL, Transact and Informix,
- .c/.h matches C, C++ and Objective-C

GUI mode: KLA detects such ambiguous situations and asks the user to select the correct technology.

CLI mode: KLA will execute by default every available engine, wasting time and resources producing confusing results. To solve this, search for the *supported.technologies* parameter when invoking KLA in CLI mode and delete the unneeded technologies.

> **Import/export SQL scripts**
>
> Export/import SQL scripts are quite common in applications, and those files are usually very large.
>
> Make sure you exclude those scripts from the analysis by changing the default SQL configuration, if you want to speed up your analysis.

Duplicated code analysis

Duplicated code analysis (aka clone detection) is also quite an intensive memory and CPU-draining task.

However, it can be configured to modify its working mode, reducing time and memory requirements.

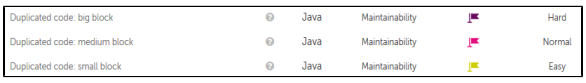
> If you are not interested at all in duplication code analysis, you can make Kiuwan not execute it:
>
> - In KLA CLI mode, specify *ignore=clones*

Kiuwan's clone detector searches for fragments of tokens that are very similar.

The term 'token' refers to each of the atomic elements identified by the analyzer. There are three types of tokens:

1. **Operators** and reserved words (specific for each language)
2. **Identifiers**: variable names, function names, etc.
3. **Literals**: numbers and string constants used in the code.

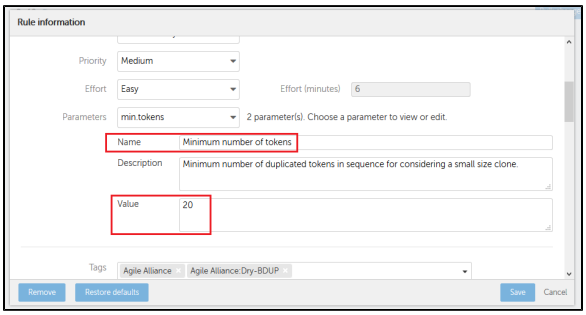
Kiuwan also generates defects of 'duplicated code' according to the size of the fragments found:



You can configure the minimum tokens that Kiuwan uses to detect a clone. This is done at two levels:

a) In Kiuwan's Local Analyzer go to **Advanced** options, then configure the number of tokens to detect a clone. You can configure a different number of clones for each language.

b) In your model, configure the minimum tokens to generate a ‘Duplicated code’ defect.



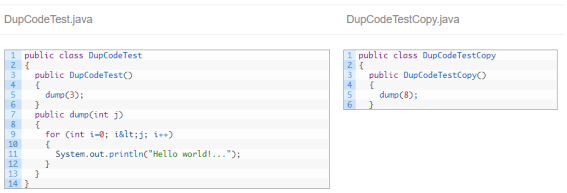
Depending on this configuration, Kiuwan gets different results in the clone detection. Let see these in detail.

The most conservative way

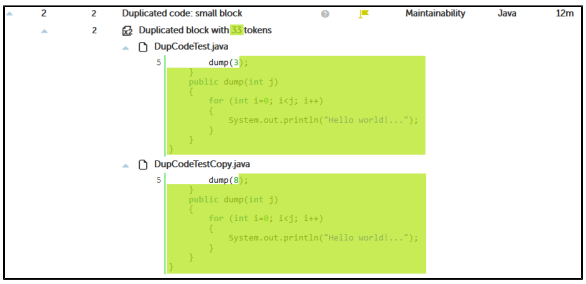
In this case, we configure Kiuwan to look for an exact match between the differents fragments:

{language}.min.tokens=20
{language}.ignore.literals=false
{language}.ignore.identifiers=false

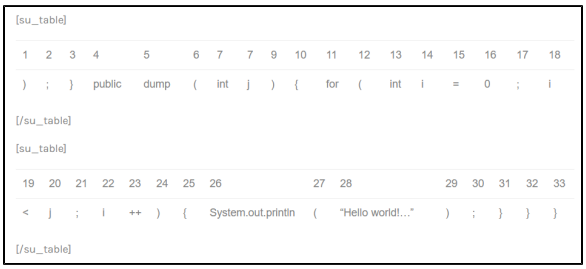
Taking this source code as example:



Kiuwan detects duplicate code:



IMPORTANT: The ‘clone’ begins at the close parenthesis of line 5, but Kiuwan prints the complete line. This may be a little messy sometimes.
The detected tokens are:


[su_table]

1	2	3	4	5	6	7	7	9	10	11	12	13	14	15	16	17	18
)	;	}	public	dump	(	int	j	)	{	for	(	int	i	=	0	;	i

[/su_table]

[su_table]

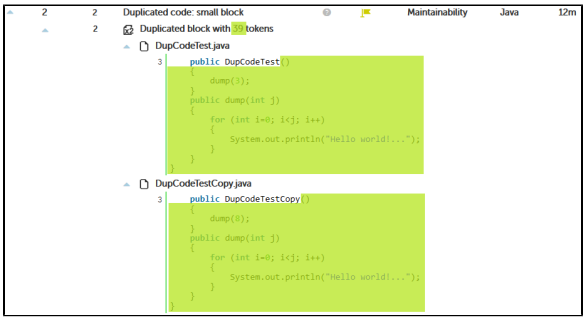
19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
<	j	;	i	++	)	{	System.out.println	(	“Hello world!...”	)	;	}	}	}

[/su_table]

Set the clone detector to be smarter

Now we are going to configure Kiuwan to ignore the numbers and string constants in our code:

{language}.min.tokens=20
{language}.ignore.literals=**true**
{language}.ignore.identifiers=false

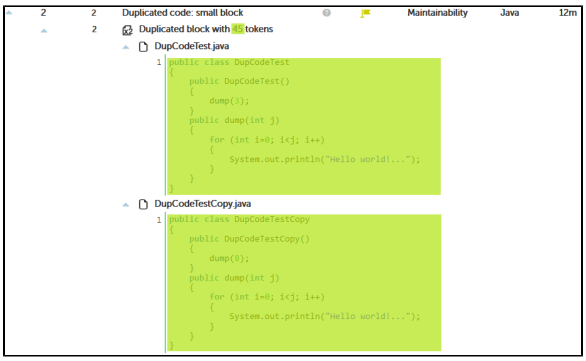


As you can see in the picture above, now the fragment is bigger because the literals (‘3’ and ‘8’) are not taken into account.

The third option

As the third option, we can ignore literals and identifiers:

{language}.min.tokens=20
{language}.ignore.literals=**true**
{language}.ignore.identifiers=**true**



With this last option, the most efficient one, it was clear that class DupCodeTestCopy is really a copy-paste where the class was only renamed, so Kiuwan detects the whole class as a clone.

But this configuration is also the one most prone to false positives. For example:

Both files have a similar structure, but functionally they are very different. Ignoring literals and identifiers, Kiuwan considers both a clone: