Java static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your JAVA code

Empty statements should be removed
Code Smell

Class variable fields should not have public accessibility
Code Smell

URIs should not be hardcoded
Code Smell

Class names should comply with a naming convention
Code Smell

Method names should comply with a naming convention
Code Smell

Comma-separated labels should be used in Switch with colon case
Code Smell

JUnit5 test classes and methods should have default package visibility
Code Smell

Track uses of "TODO" tags
Code Smell

Deprecated code should be removed
Code Smell

Annotated Mockito objects should be initialized
Bug

Custom resources should be closed
Bug

Threads should not be started in constructors
Code Smell

Regular expressions should not be too complicated

Analyze your code

Code Smell Major regex

Overly complicated regular expressions are hard to read and to maintain and can easily cause hard-to-find bugs. If a regex is too complicated, you should consider replacing it or parts of it with regular code or splitting it apart into multiple patterns at least.

The complexity of a regular expression is determined as follows:

Each of the following operators increases the complexity by an amount equal to the current nesting level and also increases the current nesting level by one for its arguments:

- `|` - when multiple `|` operators are used together, the subsequent ones only increase the complexity by 1
- `&&` (inside character classes) - when multiple `&&` operators are used together, the subsequent ones only increase the complexity by 1
- Quantifiers (`*`, `+`, `?`, `{n,m}`, `{n,}` or `{n}`)
- Non-capturing groups that set flags (such as `(?i:some_pattern)` or `(?i)some_pattern`)
- Lookahead and lookbehind assertions

Additionally, each use of the following features increase the complexity by 1 regardless of nesting:

- character classes
- back references

If a regular expression is split among multiple variables, the complexity is calculated for each variable individually, not for the whole regular expression. If a regular expression is split over multiple lines, each line is treated individually if it is accompanied by a comment (either a Java comment or a comment within the regular expression), otherwise the regular expression is analyzed as a whole.

Noncompliant Code Example

```
if (dateString.matches("^(?:(?:31(\\/|-|\\.)(?:0?[13578]|1[0
    handleDate(dateString);
}
```

Compliant Solution

```
if (dateString.matches("^\\d{1,2}([-/.])\\d{1,2}\\1\\d{1,4}$
    String dateParts[] = dateString.split("[-/.]");
    int day = Integer.parseInt(dateParts[0]);
    int month = Integer.parseInt(dateParts[1]);
    int year = Integer.parseInt(dateParts[2]);
    // Put logic to validate and process the date based on i
}
```

Exceptions

Regular expressions are only analyzed if all parts of the regular expression are either string literals, effectively final local variables or `static final` fields, all of which can be combined using the '+' operator.

When a regular expression is split among multiple variables or commented lines, each part is only analyzed if it is syntactically valid by itself.

Available In:

sonarlint sonarcloud sonarqube

"main" should not "throw" anything

Code Smell

Track lack of copyright and license headers

Code Smell

Octal values should not be used

Code Smell

Exit methods should not be called

Code Smell

HTTP response headers should not be vulnerable to injection attacks