'PRO2PROJECT'

VERSION 4.2.0

CODE ANALYSIS

By: default

2023-08-09

'pro2project'

CONTENT

'pro2project'

INTRODUCTION

This document contains results of the code analysis of 'pro2project'.

CNES app/plugin for SonarQube that allows users to export analysis reports as OpenXML, Markdown and CSV.

CONFIGURATION

- Quality Profiles
  - Names: Sonar way [Java];
  - Files: AYlzSd28f_Qmj1ZxJvYY.json;
- Quality Gate
  - Name: AYlEo0Ql0S8MdK3ohUdJ
  - File: AYlEo0Ql0S8MdK3ohUdJ.xml

'pro2project'

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability

QUALITY GATE STATUS

Quality Gate Status

Metric	Value
Reliability Rating on New Code	OK
Security Rating on New Code	OK
Maintainability Rating on New Code	OK

METRICS

Coverage	Duplica…	…omment density	Median number of lines of code per file	Adherence to coding standard
0.0 %	0.6 %	31.4 %	41.0	99.5 %

TESTS

Total	Success Rate	Skipped	Errors	Failures
0	0 %	0	0	0

'pro2project'

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
0d 0h 15min	-	0d 3h 0min	**0d 3h 15min**

'pro2project'

METRICS RANGE

	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	1.0	0.0	0.0	0.0
Max	919.0	471.0	5995.0	75.0	0.0	34.0

VOLUME

Language	Number
Java	5995
Total	5995

'pro2project'

ISSUES

CHARTS

ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	0	1	0	0
VULNERABILITY	0	0	0	0	0
CODE_SMELL	0	4	8	4	0

ISSUES LIST

Name	Description	Type	Severity	Number
Conditionally executed code should be reachable	Why is this an issue?Conditional expressions which are always true or false can lead to dead code. Such code is always buggy and should never be used in production. Noncompliant code examplea = false;if (a) { // Noncompliant doSomething(); // never executed}if (!a | { // Noncompliant; "!a" is always "true", "b" is never	BUG	MAJOR	1

'pro2project'

	evaluated doSomething(); } else { doSomethingElse(); // never executed }Exceptions This rule will not raise an issue either of these cases: When the condition is a single final boolean final boolean debug = false;//...if (debug) { // something} When the condition is literally true or false. (true) { // do something}In these cases it is obvious the code is as intended.Resources MITRE, CWE-570 - Expression is Always False MITRE, CWE-571 - Expression is Always True CERT, MSC12-C. - Detect and remove code that has no effect or is never executed			
String literals should not be duplicated	Why is this an issue?Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.On the other hand, constants can be referenced from many places, but only need to be updated in a single place.Noncompliant code exampleWith the default threshold of 3:public void run() { prepare("action1"); // Noncompliant - "action1" is duplicated 3 times execute("action1"); release("action1");}@SuppressWarning("all") // Compliant - annotations are excludedprivate void method1() { /* ... */ }@SuppressWarning("all")private void method2() { /* ... */ }public String method3(String a) { System.out.println("'" + a + "'"); // Compliant - literal "'" has less than 5 characters and is excluded return ""; // Compliant - literal "" has less than 5 characters and is excluded}Compliant solutionprivate static final String ACTION_1 = "action1"; // Compliantpublic void run() { prepare(ACTION_1); // Compliant execute(ACTION_1); release(ACTION_1);}Exceptions To prevent generating some false-positives, literals having less than 5 characters are excluded.	CODE_SMELL	CRITICAL	1
Cognitive Complexity of methods should not be too high	Why is this an issue?Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.Exceptionsequals and hashCode methods are ignored because they might be automatically generated and might end up being difficult to understand, especially in presence of many fields.Resources Cognitive Complexity	CODE_SMELL	CRITICAL	3
Unused "private" fields should be removed	Why is this an issue?If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for.Note that this rule does not take reflection into account, which means that issues will be raised on private fields that are only accessed using the reflection API.Noncompliant code examplepublic class	CODE_SMELL	MAJOR	1

'pro2project'

	MyClass {□ private int foo = 42;□□ public int compute(int a) { return a * 42;□ }□□}□□Compliant solution□□public class MyCla public int compute(int a) {□ return a * 42;□ }□}□□Exceptions rule admits 3 exceptions:□□ Serialization id fields □□The Java serialization runtime associates with each serializable class a version number, called serialVersionUID, which is used during□deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to□serialization.□A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that □must be static, final, and of type long. By definition those serialVersionUID fields should not be reported by this rule: □public class MyClass implements java.io.Serializable { private static final long serialVersionUID = 42L;□}□□□ Annotat fields □□The unused field in this class will not be reported by the rule as it is annotated.□□public class MyClass { @SomeAnnotation□ private int unused;□}□□□ Fields from cla with native methods □□The unused field in this class will not be reported by the rule as it might be used by native code. □public class MyClass {□ private int unused = 42;□ private native static void doSomethingNative();□}			
Methods should not have too many parameters	Why is this an issue?□A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many□things. □Noncompliant code example□With a maximum number of 4 parameters:□□public void doSomething(int param1, int param2, int param3, String param4, long param5) {□...□} □Compliant solution□□public void doSomething(int param1, ir param2, int param3, String param4) {□...□}□□Exceptions□Meth annotated with :□□ Spring's @RequestMapping (and related shortcut annotations, like @GetRequest) □ JAX-RS API annotations (like @javax.ws.rs.GET) □ Bean constructor injection with @org.springframework.beans.factory.annotation.Autowired □ CDI constructor injection with @javax.inject.Inject @com.fasterxml.jackson.annotation.JsonCreator Micronaut's annotations (like @io.micronaut.http.annotation.Get) □□may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored.□Also, if a class annotated as a Spring component (like @org.springframework.stereotype.Component) has a single constructor, that□constructor will be considered @Autowired and ignored by the rule.	CODE_SMELL	MAJOR	1
Sections of code should not be commented out	Why is this an issue?□Programmers should not comment out code as it bloats programs and reduces readability.□Unused code should be deleted and can be retrieved from source	CODE_SMELL	MAJOR	2

'pro2project'

	control history if required.			
Unused assignments should be removed	Why is this an issue?A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a valueonly to then overwrite it or throw it away, could indicate a serious error in the code. Even if it’s not an error, it is at best a waste of resources.Therefore all calculated values should be used.Noncompliant code exampleei = a + b; // Noncompliant; calculation result not used before value is overwrittenei = compute();eCompliant solutionei = a + b;ei compute();eExceptionsThis rule ignores initializations to -1, 1, null, true, false and "".eResources MITRE, CWE-563 - Assignment to Variable without Use ('Unused Variable') CERT, MSC13-C. - Detect and remove unused values CERT, MSC56-J. - Detect and remove superfluous code and values	CODE_SMELL	MAJOR	4
Unused local variables should be removed	Why is this an issue?If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable is used for.Noncompliant code example public int numberOfMinutes(int hours) { int seconds = 0; // seconds is never used return hours * 60;}Compliant solutionpublic int numberOfMinutes(int hours) { return hours * 60;}	CODE_SMELL	MINOR	3
"@Deprecated" code should not be used	Why is this an issue?Once deprecated, classes, and interfaces, and their members should be avoided, rather than used, inherited or extended. Deprecation is a warning that the class or interface has been superseded, and will eventually be removed. The deprecation period allows you to make a smooth transition awayfrom the aging, soon-to-be-retired technology.Noncompliant code example/** * @deprecated As of release 1.3, replaced by {@link #Fee} */ @Deprecatedpublic class Fum { ... }public class Foo { /** @deprecated As of release 1.7, replaced by {@link #doTheThingBetter()} */ @Deprecated public void doTheThing() { ... } public void doTheThingBetter() { ... }} public class Bar extends Foo { public void doTheThing() { ... // Noncompliant; don't override a deprecated method or explicitly mark it as @Deprecated}public class Bar extends Fum { // Noncompliant; Fum is deprecated public void myMethod() { Foo foo = new Foo(); // okay; the class isn't deprecated foo.doTheThing(); // Noncompliant; doTheThing method is deprecated }}eResources MITR CWE-477 - Use of Obsolete Functions CERT, MET02-J. - Do not use deprecated or obsolete classes or methods	CODE_SMELL	MINOR	1

'pro2project'

'pro2project'

SECURITY HOTSPOTS

SECURITY HOTSPOTS COUNT BY CATEGORY AND PRIORITY

Category / Priority	LOW	MEDIUM	HIGH
LDAP Injection	0	0	0
Object Injection	0	0	0
Server-Side Request Forgery (SSRF)	0	0	0
XML External Entity (XXE)	0	0	0
Insecure Configuration	0	0	0
XPath Injection	0	0	0
Authentication	0	0	0
Weak Cryptography	0	0	0
Denial of Service (DoS)	0	0	0
Log Injection	1	0	0
Cross-Site Request Forgery (CSRF)	0	0	0
Open Redirect	0	0	0
Permission	0	0	0
SQL Injection	0	0	0
Encryption of Sensitive Data	0	0	0
Traceability	0	0	0
Buffer Overflow	0	0	0
File Manipulation	0	0	0
Code Injection (RCE)	0	0	0

'pro2project'

Cross-Site Scripting (XSS)	0	0	0
Command Injection	0	0	0
Path Traversal Injection	0	0	0
HTTP Response Splitting	0	0	0
Others	1	0	0

SECURITY HOTSPOTS LIST

Category	Name	Priority	Severity	Count
Log Injection	Configuring loggers is security-sensitive	LOW	CRITICAL	1
Others	Using publicly writable directories is security-sensitive	LOW	CRITICAL	1