CRP-GATEWAY-PARENT

VERSION 1.9.6-CUB

Code analysis

By: Administrator

2024-10-07

CONTENT

crp-gateway-parent

INTRODUCTION

This document contains results of the code analysis of crp-gateway-parent.

Parent pom providing dependency and plugin management for applications built with Maven

CONFIGURATION

- Quality Profiles
  - Names: Sonar way [Java]; Sonar way [XML];
  - Files: AYEqSPBlRapRvVRDGywn.json; AYEqSPPaRapRvVRDGy6D.json;
- Quality Gate
  - Name: Standard
  - File: Standard.xml

crp-gateway-parent

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability
A	A	A	A

QUALITY GATE STATUS

Quality Gate Status	Passed

Metric	Value
Reliability Rating on New Code	OK
Security Rating on New Code	OK
Maintainability Rating on New Code	OK
Duplicated Lines (%) on New Code	OK
Security Hotspots Reviewed on New Code	OK

METRICS

Coverage	Duplication	Comment density	Median number of lines of code per file	Adherence to coding standard
0.0 %	4.1 %	6.3 %	47.0	97.3 %

TESTS

Total	Success Rate	Skipped	Errors	Failures

crp-gateway-parent

0	**0 %**	**0**	**0**	**0**

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
-	-	8d 6h 15min	**8d 6h 15min**

crp-gateway-parent

METRICS RANGE

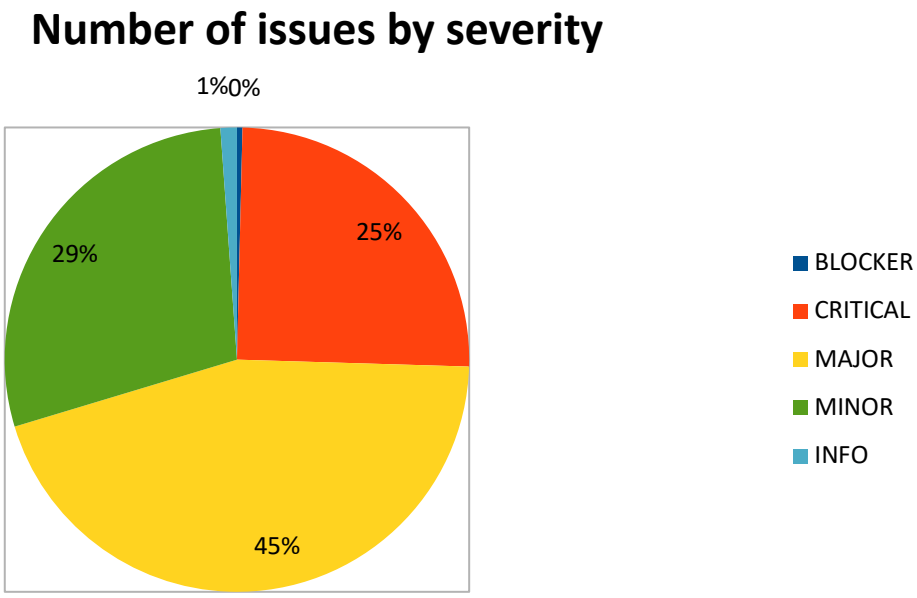
	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	3.0	0.0	0.0	0.0
Max	2486.0	1698.0	18604.0	62.1	0.0	61.7

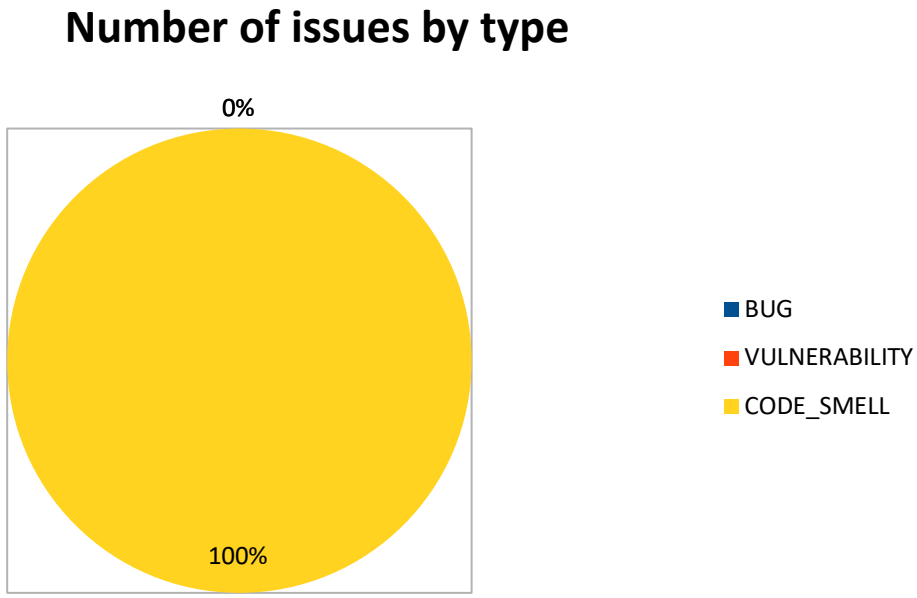
VOLUME

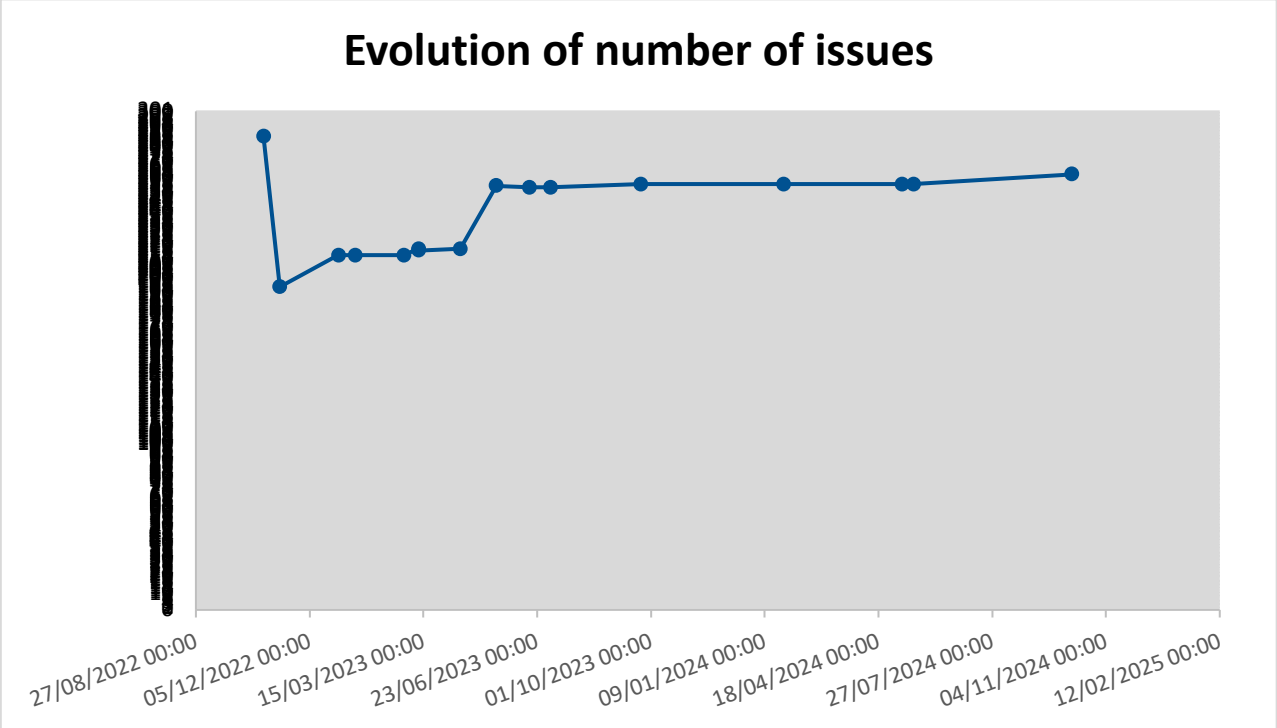
Language	Number
Java	35332
XML	2190
Total	37522

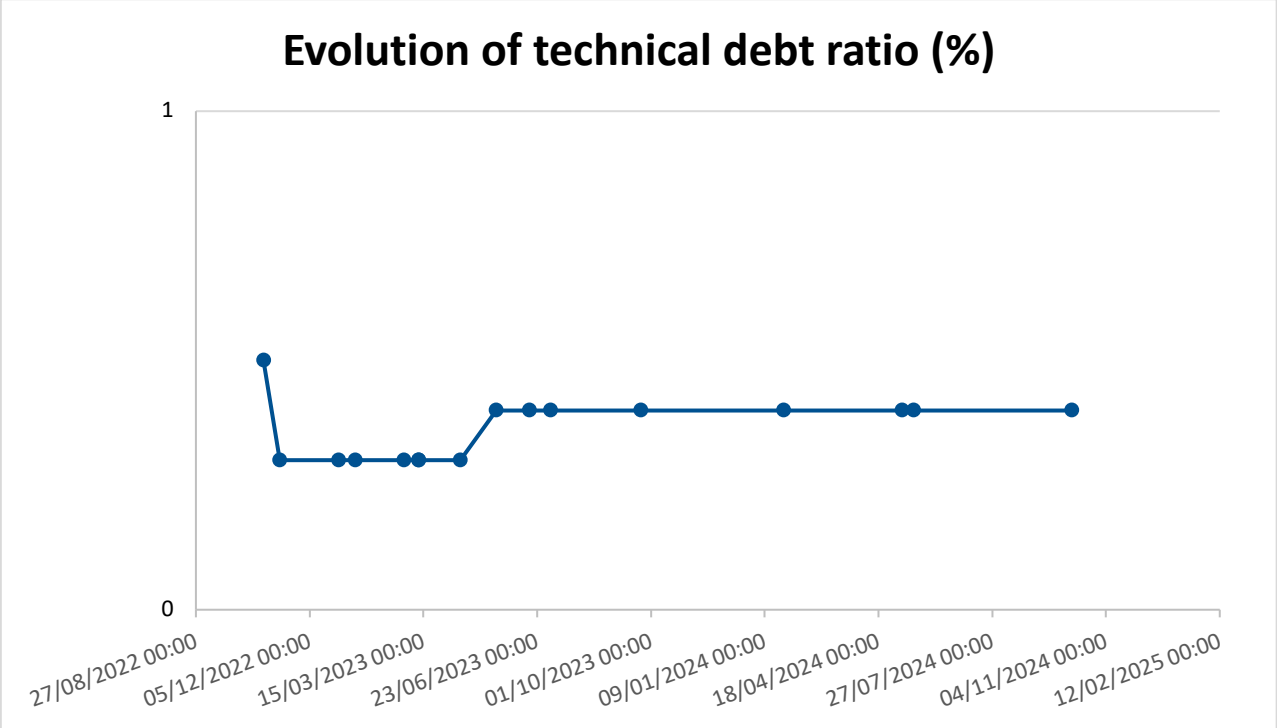
ISSUES

CHARTS






Evolution of number of issues
27/08/2022 00:00
05/12/2022 00:00
15/03/2023 00:00
23/06/2023 00:00
01/10/2023 00:00
09/01/2024 00:00
18/04/2024 00:00
27/07/2024 00:00
04/11/2024 00:00
12/02/2025 00:00


Evolution of technical debt ratio (%)
1
0
27/08/2022 00:00
05/12/2022 00:00
15/03/2023 00:00
23/06/2023 00:00
01/10/2023 00:00
09/01/2024 00:00
18/04/2024 00:00
27/07/2024 00:00
04/11/2024 00:00
12/02/2025 00:00

crp-gateway-parent

ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	0	0	0	0
VULNERABILITY	0	0	0	0	0
CODE_SMELL	3	75	118	66	1

ISSUES LIST

Name	Description	Type	Severity	Number
Child class fields should not shadow parent class fields	Having a variable with the same name in two unrelated classes is fine, but do the same thing within a class hierarchy and you'll get confusion at best, chaos at worst. Noncompliant Code Example public class Fruit { protected Season ripe; protected Color flesh; // ... } public class Raspberry extends Fruit { private boolean ripe; // Noncompliant private static Color FLESH; // Noncompliant } Compliant Solution public class Fruit { protected Season ripe; protected Color flesh; // ... } public class Raspberry extends Fruit { private boolean ripened; private static Color FLESH_COLOR; } Exceptions This rule ignores same-name fields that are static in both the parent and child classes. This rule ignores private parent class fields, but in all other such cases, the child class field should be renamed. public class Fruit { private Season ripe; // ... } public class Raspberry extends Fruit { private Season ripe; // Compliant as parent field 'ripe' is anyway not visible from Raspberry // ... }	CODE_SMELL	BLOCKER	1
Methods should not be empty	There are several reasons for a method not to have a method body: It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production. It is not yet, or never will be, supported. In this case an UnsupportedOperationException should be thrown. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example public void doSomething() { } public void doSomethingElse() { } Compliant Solution @Override public void doSomething() { // Do nothing because of X and Y. } @Override public void	CODE_SMELL	CRITICAL	11

	doSomethingElse() { throw new UnsupportedOperationException(); } Exceptions Default (no-argument) constructors are ignored when there are other constructors in the class, as are empty methods in abstract classes. public abstract class Animal { void speak() { // default implementation ignored } }			
String literals should not be duplicated	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 Example With 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 excluded private 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 Solution private static final String ACTION_1 = "action1"; // Compliant public 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	21
Generic wildcard types should not be used in return types	It is highly recommended not to use wildcard types as return types. Because the type inference rules are fairly complex it is unlikely the user of that API will know how to use it correctly. Let’s take the example of method returning a "List<? extends Animal>". Is it possible on this list to add a Dog, a Cat, … we simply don’t know. And neither does the compiler, which is why it will not allow such a direct use. The use of wildcard types should be limited to method parameters. This rule raises an issue when a method returns a wildcard type. Noncompliant Code Example List<? extends Animal> getAnimals(){...} Compliant Solution List<Animal> getAnimals(){...} or List<Dog> getAnimals(){...}	CODE_SMELL	CRITICAL	10

Fields in a "Serializable" class should either be transient or serializable	Fields in a Serializable class must themselves be either Serializable or transient even if the class is never explicitly serialized or deserialized. For instance, under load, most J2EE application frameworks flush objects to disk, and an allegedly Serializable object with non-transient, non-serializable data members could cause program crashes, and open the door to attackers. In general a Serializable class is expected to fulfil its contract and not have an unexpected behaviour when an instance is serialized. This rule raises an issue on non-Serializable fields, and on collection fields when they are not private (because they could be assigned non-Serializable values externally), and when they are assigned non-Serializable types within the class. Noncompliant Code Example public class Address { //... } public class Person implements Serializable { private static final long serialVersionUID = 1905122041950251207L; private String name; private Address address; // Noncompliant; Address isn't serializable } Compliant Solution public class Address implements Serializable { private static final long serialVersionUID = 2405172041950251807L; } public class Person implements Serializable { private static final long serialVersionUID = 1905122041950251207L; private String name; private Address address; } Exceptions The alternative to making all members serializable or transient is to implement special methods which take on the responsibility of properly serializing and de-serializing the object. This rule ignores classes which implement the following methods: private void writeObject(java.io.ObjectOutputStream out) throws IOException private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException; See MITRE, CWE-594 - Saving Unserializable Objects to Disk Oracle Java 6, Serializable Oracle Java 7, Serializable	CODE_SMELL	CRITICAL	1
Instance methods should not write to "static" fields	Correctly updating a static field from a non-static method is tricky to get right and could easily lead to bugs if there are multiple class instances and/or multiple threads in play. Ideally, static fields are only updated from synchronized static methods. This rule raises an issue each time a static field is updated from a non-static method. Noncompliant Code Example public class MyClass { private static int count = 0; public void doSomething() { //... count++; //	CODE_SMELL	CRITICAL	1

	Noncompliant } }			
Cognitive Complexity of methods should not be too high	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. See Cognitive Complexity	CODE_SMELL	CRITICAL	20
A conditionally executed single line should be denoted by indentation	In the absence of enclosing curly braces, the line immediately after a conditional is the one that is conditionally executed. By both convention and good practice, such lines are indented. In the absence of both curly braces and indentation the intent of the original programmer is entirely unclear and perhaps not actually what is executed. Additionally, such code is highly likely to be confusing to maintainers. Noncompliant Code Example if (condition) // Noncompliant doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); Compliant Solution if (condition) doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo();	CODE_SMELL	CRITICAL	1
"String#replace" should be preferred to "String#replaceAll"	The underlying implementation of String::replaceAll calls the java.util.regex.Pattern.compile() method each time it is called even if the first argument is not a regular expression. This has a significant performance cost and therefore should be used with care. When String::replaceAll is used, the first argument should be a real regular expression. If it's not the case, String::replace does exactly the same thing as String::replaceAll without the performance drawback of the regex. This rule raises an issue for each String::replaceAll used with a String as first parameter which doesn't contains special regex character or pattern. Noncompliant Code Example String init = "Bob is a Bird... Bob is a Plane... Bob is Superman!"; String changed = init.replaceAll("Bob is", "It's"); // Noncompliant changed = changed.replaceAll("\\.\\.\\.", ";"); // Noncompliant Compliant Solution String init = "Bob is a Bird... Bob is a Plane... Bob is Superman!"; String changed = init.replace("Bob is", "It's"); changed = changed.replace("...", ";"); Or, with a regex: String init = "Bob is a Bird... Bob is a Plane... Bob is Superman!"; String changed = init.replaceAll("\\w*\\sis", "It's"); changed = changed.replaceAll("\\.{3}", ";"); See S4248 - Regex patterns should not be created needlessly	CODE_SMELL	CRITICAL	1

Track uses of "TODO" tags	TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example void doSomething() { // TODO } See MITRE, CWE-546 - Suspicious Comment	CODE_SMELL	INFO	3
Source files should not have any duplicated blocks	An issue is created on a file as soon as there is at least one block of duplicated code on this file	CODE_SMELL	MAJOR	36
Standard outputs should not be used directly to log anything	When logging a message there are several important requirements which must be fulfilled: The user must be able to easily retrieve the logs The format of all logged message must be uniform to allow the user to easily read the log Logged data must actually be recorded Sensitive data must only be logged securely If a program directly writes to the standard outputs, there is absolutely no way to comply with those requirements. That's why defining and using a dedicated logger is highly recommended. Noncompliant Code Example System.out.println("My Message"); // Noncompliant Compliant Solution logger.log("My Message"); See OWASP Top 10 2021 Category A9 - Security Logging and Monitoring Failures OWASP Top 10 2017 Category A3 - Sensitive Data Exposure CERT, ERR02-J. - Prevent exceptions while logging data	CODE_SMELL	MAJOR	3
Collapsible "if" statements should be merged	Merging collapsible if statements increases the code's readability. Noncompliant Code Example if (file != null) { if (file.isFile() || file.isDirectory()) { /* ... */ } } Compliant Solution if (file != null && isFileOrDirectory(file)) { /* ... */ } private static boolean isFileOrDirectory(File file) { return file.isFile() || file.isDirectory(); }	CODE_SMELL	MAJOR	5
Local variables should not shadow class fields	Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they're using one variable but are really using another. Noncompliant Code Example class Foo { public int myField; public void doSomething() { int myField = 0; ... } } See	CODE_SMELL	MAJOR	6

	CERT, DCL01-C. - Do not reuse variable names in subscopes CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes			
Utility classes should not have public constructors	Utility classes, which are collections of static members, are not meant to be instantiated. Even abstract utility classes, which can be extended, should not have public constructors. Java adds an implicit public constructor to every class which does not define at least one explicitly. Hence, at least one non-public constructor should be defined. Noncompliant Code Example class StringUtils { // Noncompliant public static String concatenate(String s1, String s2) { return s1 + s2; } } Compliant Solution class StringUtils { // Compliant private StringUtils() { throw new IllegalStateException("Utility class"); } public static String concatenate(String s1, String s2) { return s1 + s2; } } Exceptions When class contains public static void main(String[] args) method it is not considered as utility class and will be ignored by this rule.	CODE_SMELL	MAJOR	11
Generic exceptions should never be thrown	Using such generic exceptions as Error, RuntimeException, Throwable, and Exception prevents calling methods from handling true, system-generated exceptions differently than application-generated errors. Noncompliant Code Example public void foo(String bar) throws Throwable { // Noncompliant throw new RuntimeException("My Message"); // Noncompliant } Compliant Solution public void foo(String bar) { throw new MyOwnRuntimeException("My Message"); } Exceptions Generic exceptions in the signatures of overriding methods are ignored, because overriding method has to follow signature of the throw declaration in the superclass. The issue will be raised on superclass declaration of the method (or won't be raised at all if superclass is not part of the analysis). @Override public void myMethod() throws Exception {...} Generic exceptions are also ignored in the signatures of methods that make calls to methods that throw generic exceptions. public void myOtherMethod throws Exception { doTheThing(); // this method throws Exception } See MITRE, CWE-397 - Declaration of Throws for Generic Exception CERT, ERR07-J. - Do not throw RuntimeException, Exception, or Throwable	CODE_SMELL	MAJOR	8

Try-catch blocks should not be nested	Nesting try/catch blocks severely impacts the readability of source code because it makes it too difficult to understand which block will catch which exception.	CODE_SMELL	MAJOR	11
Empty arrays and collections should be returned instead of null	Returning null instead of an actual array, collection or map forces callers of the method to explicitly test for nullity, making them more complex and less readable. Moreover, in many cases, null is used as a synonym for empty. Noncompliant Code Example public static List<Result> getAllResults() { return null; // Noncompliant } public static Result[] getResults() { return null; // Noncompliant } public static Map<String, Object> getValues() { return null; // Noncompliant } public static void main(String[] args) { Result[] results = getResults(); if (results != null) { // Nullity test required to prevent NPE for (Result result: results) { /* ... */ } } List<Result> allResults = getAllResults(); if (allResults != null) { // Nullity test required to prevent NPE for (Result result: allResults) { /* ... */ } } Map<String, Object> values = getValues(); if (values != null) { // Nullity test required to prevent NPE values.forEach((k, v) -> doSomething(k, v)); } } Compliant Solution public static List<Result> getAllResults() { return Collections.emptyList(); // Compliant } public static Result[] getResults() { return new Result[0]; // Compliant } public static Map<String, Object> getValues() { return Collections.emptyMap(); // Compliant } public static void main(String[] args) { for (Result result: getAllResults()) { /* ... */ } for (Result result: getResults()) { /* ... */ } getValues().forEach((k, v) -> doSomething(k, v)); } See CERT, MSC19-C. - For functions that return an array, prefer returning an empty array over a null value CERT, MET55-J. - Return an empty array or collection instead of a null value for methods that return an array or collection	CODE_SMELL	MAJOR	12
Unused method parameters should be removed	Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same. Noncompliant Code Example void doSomething(int a, int b) { // "b" is unused compute(a); } Compliant Solution void doSomething(int a) { compute(a); } Exceptions The rule will not raise issues for unused parameters: that are annotated with	CODE_SMELL	MAJOR	2

	@javax.enterprise.event.Observes in overrides and implementation methods in interface default methods in non-private methods that only throw or that have empty bodies in annotated methods, unless the annotation is @SuppressWarning("unchecked") or @SuppressWarning("rawtypes"), in which case the annotation will be ignored in overridable methods (non-final, or not member of a final class, non-static, non-private), if the parameter is documented with a proper javadoc. @Override void doSomething(int a, int b) { // no issue reported on b compute(a); } public void foo(String s) { // designed to be extended but noop in standard case } protected void bar(String s) { //open-closed principle } public void qix(String s) { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } /** * @param s This string may be use for further computation in overriding classes */ protected void foobar(int a, String s) { // no issue, method is overridable and unused parameter has proper javadoc compute(a); } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed			
Throwable and Error should not be caught	Throwable is the superclass of all errors and exceptions in Java. Error is the superclass of all errors, which are not meant to be caught by applications. Catching either Throwable or Error will also catch OutOfMemoryError and InternalError, from which an application should not attempt to recover. Noncompliant Code Example try { /* ... */ } catch (Throwable t) { /* ... */ } try { /* ... */ } catch (Error e) { /* ... */ } Compliant Solution try { /* ... */ } catch (RuntimeException e) { /* ... */ } try { /* ... */ } catch (MyException e) { /* ... */ } See MITRE, CWE-396 - Declaration of Catch for Generic Exception C++ Core Guidelines E.14 - Use purpose-designed user-defined types as exceptions (not built-in types)	CODE_SMELL	MAJOR	7
Sections of code should not be commented out	Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required.	CODE_SMELL	MAJOR	2
Anonymous inner classes containing only one method should	Before Java 8, the only way to partially support closures in Java was by using anonymous inner classes. But the syntax of anonymous classes may seem unwieldy and unclear. With Java 8, most uses of	CODE_SMELL	MAJOR	1

become lambdas	anonymous inner classes should be replaced by lambdas to highly increase the readability of the source code. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 8. Noncompliant Code Example myCollection.stream().map(new Mapper<String,String>() { public String map(String input) { return new StringBuilder(input).reverse().toString(); } }); Predicate<String> isEmpty = new Predicate<String> { boolean test(String myString) { return myString.isEmpty(); } } Compliant Solution myCollection.stream().map(input -> new StringBuilder(input).reverse().toString()); Predicate<String> isEmpty = myString -> myString.isEmpty();			
Unused assignments should be removed	A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a value only 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 Example i = a + b; // Noncompliant; calculation result not used before value is overwritten i = compute(); Compliant Solution i = a + b; i += compute(); Exceptions This rule ignores initializations to -1, 0, 1, null, true, false and "". See 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	2
Unused type parameters should be removed	Type parameters that aren't used are dead code, which can only distract and possibly confuse developers during maintenance. Therefore, unused type parameters should be removed. Noncompliant Code Example int <T> Add(int a, int b) // Noncompliant; <T> is ignored { return a + b; } Compliant Solution int Add(int a, int b) { return a + b; }	CODE_SMELL	MAJOR	2
Boolean expressions should not be gratuitous	If a boolean expression doesn't change the evaluation of the condition, then it is entirely unnecessary, and can be removed. If it is gratuitous because it does not match the programmer's intent, then it's a bug and the expression should be fixed. Noncompliant Code Example a = true; if (a) { // Noncompliant	CODE_SMELL	MAJOR	3

	doSomething(); } if (b && a) { // Noncompliant; "a" is always "true" doSomething(); } if (c || !a) { // Noncompliant; "!a" is always "false" doSomething(); } Compliant Solution a = true; if (foo(a)) { doSomething(); } if (b) { doSomething(); } if (c) { doSomething(); } See MITRE, CWE-571 - Expression is Always True MITRE, CWE-570 - Expression is Always False			
Printf-style format strings should be used correctly	Because printf-style format strings are interpreted at runtime, rather than validated by the compiler, they can contain errors that result in the wrong strings being created. This rule statically validates the correlation of printf-style format strings to their arguments when calling the format(...) methods of java.util.Formatter, java.lang.String, java.io.PrintStream, MessageFormat, and java.io.PrintWriter classes and the printf(...) methods of java.io.PrintStream or java.io.PrintWriter classes. Noncompliant Code Example String.format("First {0} and then {1}", "foo", "bar"); //Noncompliant. Looks like there is a confusion with the use of {{java.text.MessageFormat}}, parameters "foo" and "bar" will be simply ignored here String.format("Display %3$d and then %d", 1, 2, 3); //Noncompliant; the second argument '2' is unused String.format("Too many arguments %d and %d", 1, 2, 3); //Noncompliant; the third argument '3' is unused String.format("First Line\n"); //Noncompliant; %n should be used in place of \n to produce the platform-specific line separator String.format("Is myObject null ? %b", myObject); //Noncompliant; when a non-boolean argument is formatted with %b, it prints true for any nonnull value, and false for null. Even if intended, this is misleading. It's better to directly inject the boolean value (myObject == null in this case) String.format("value is " + value); // Noncompliant String s = String.format("string without arguments"); // Noncompliant MessageFormat.format("Result '{0}'.", value); // Noncompliant; String contains no format specifiers. (quote are discarding format specifiers) MessageFormat.format("Result {0}.", value, value); // Noncompliant; 2nd argument is not used MessageFormat.format("Result {0}.", myObject.toString()); // Noncompliant; no need to call toString() on objects java.util.Logger logger; logger.log(java.util.logging.Level.SEVERE, "Result {0}.", myObject.toString()); // Noncompliant; no need to call toString() on objects	CODE_SMELL	MAJOR	2

	logger.log(java.util.logging.Level.SEVERE, "Result.", new Exception()); // compliant, parameter is an exception logger.log(java.util.logging.Level.SEVERE, "Result '{0}'", 14); // Noncompliant - String contains no format specifiers. logger.log(java.util.logging.Level.SEVERE, "Result " + param, exception); // Noncompliant; Lambda should be used to differ string concatenation. org.slf4j.Logger slf4jLog; org.slf4j.Marker marker; slf4jLog.debug(marker, "message {}"); slf4jLog.debug(marker, "message", 1); // Noncompliant - String contains no format specifiers. org.apache.logging.log4j.Logger log4jLog; log4jLog.debug("message", 1); // Noncompliant - String contains no format specifiers. Compliant Solution String.format("First %s and then %s", "foo", "bar"); String.format("Display %2$d and then %d", 1, 3); String.format("Too many arguments %d %d", 1, 2); String.format("First Line%n"); String.format("Is myObject null ? %b", myObject == null); String.format("value is %d", value); String s = "string without arguments"; MessageFormat.format("Result {0}.", value); MessageFormat.format("Result '{0}' = {0}", value); MessageFormat.format("Result {0}.", myObject); java.util.Logger logger; logger.log(java.util.logging.Level.SEVERE, "Result {0}.", myObject); logger.log(java.util.logging.Level.SEVERE, "Result {0}'", 14); logger.log(java.util.logging.Level.SEVERE, exception, () -> "Result " + param); org.slf4j.Logger slf4jLog; org.slf4j.Marker marker; slf4jLog.debug(marker, "message {}"); slf4jLog.debug(marker, "message {}", 1); org.apache.logging.log4j.Logger log4jLog; log4jLog.debug("message {}", 1); See CERT, FIO47-C. - Use valid format strings			
Assignments should not be redundant	The transitive property says that if a == b and b == c, then a == c. In such cases, there’s no point in assigning a to c or vice versa because they’re already equivalent. This rule raises an issue when an assignment is useless because the assigned-to variable already holds the value on all execution paths. Noncompliant Code Example a = b; c = a; b = c; // Noncompliant: c and b are already the same Compliant Solution a = b; c = a;	CODE_SMELL	MAJOR	1
Restricted Identifiers should not be used as	Even if it is technically possible, Restricted Identifiers should not be used as identifiers. This is only possible	CODE_SMELL	MAJOR	4

Identifiers	for compatibility reasons, using it in Java code is confusing and should be avoided. Note that this applies to any version of Java, including the one where these identifiers are not yet restricted, to avoid future confusion. This rule reports an issue when restricted identifiers: var yield record are used as identifiers. Noncompliant Code Example var var = "var"; // Noncompliant: compiles but this code is confusing var = "what is this?"; int yield(int i) { // Noncompliant return switch (i) { case 1: yield(0); // This is a yield from switch expression, not a recursive call. default: yield(i-1); }; } String record = "record"; // Noncompliant Compliant Solution var myVariable = "var"; int minusOne(int i) { return switch (i) { case 1: yield(0); default: yield(i-1); }; } String myRecord = "record"; See JLS16, 3.8: Identifiers			
Class variable fields should not have public accessibility	Public class variable fields do not respect the encapsulation principle and has three main disadvantages: Additional behavior such as validation cannot be added. The internal representation is exposed, and cannot be changed afterwards. Member values are subject to change from anywhere in the code and may not meet the programmer's assumptions. By using private attributes and accessor methods (set and get), unauthorized modifications are prevented. Noncompliant Code Example public class MyClass { public static final int SOME_CONSTANT = 0; // Compliant - constants are not checked public String firstName; // Noncompliant } Compliant Solution public class MyClass { public static final int SOME_CONSTANT = 0; // Compliant - constants are not checked private String firstName; // Compliant public String getFirstName() { return firstName; } public void setFirstName(String firstName) { this.firstName = firstName; } } Exceptions Because they are not modifiable, this rule ignores public final fields. Also, annotated fields, whatever the annotation(s) will be ignored, as annotations are often used by injection frameworks, which in exchange require having public fields. See MITRE, CWE-493 - Critical Public Variable Without Final Modifier	CODE_SMELL	MINOR	3
Empty statements should be removed	Empty statements, i.e. ;, are usually introduced by mistake, for example because: It was meant to be replaced by an actual statement, but this was	CODE_SMELL	MINOR	7

	forgotten. There was a typo which lead the semicolon to be doubled, i.e. ;;. Noncompliant Code Example void doSomething() { ; // Noncompliant - was used as a kind of TODO marker } void doSomethingElse() { System.out.println("Hello, world!");; // Noncompliant - double ; ... } Compliant Solution void doSomething() {} void doSomethingElse() { System.out.println("Hello, world!"); ... for (int i = 0; i < 3; i++) ; // compliant if unique statement of a loop ... } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed CERT, MSC51-J. - Do not place a semicolon immediately following an if, for, or while condition CERT, EXP15-C. - Do not place a semicolon on the same line as an if, for, or while statement			
Return of boolean expressions should not be wrapped into an "if-then-else" statement	Return of boolean literal statements wrapped into if-then-else ones should be simplified. Similarly, method invocations wrapped into if-then-else differing only from boolean literals should be simplified into a single invocation. Noncompliant Code Example boolean foo(Object param) { if (expression) { // Noncompliant bar(param, true, "qix"); } else { bar(param, false, "qix"); } if (expression) { // Noncompliant return true; } else { return false; } } Compliant Solution boolean foo(Object param) { bar(param, expression, "qix"); return expression; }	CODE_SMELL	MINOR	1
Unnecessary imports should be removed	The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer. Unused and useless imports should not occur if that is the case. Leaving them in reduces the code’s readability, since their presence can be confusing. Noncompliant Code Example package my.company; import java.lang.String; // Noncompliant; java.lang classes are always implicitly imported import my.company.SomeClass; // Noncompliant; same-package files are always implicitly imported import java.io.File; // Noncompliant; File is not used import my.company2.SomeType; import my.company2.SomeType; // Noncompliant; 'SomeType' is already imported class ExampleClass { public String someString; public SomeType something; } Exceptions Imports for types mentioned in Javadocs are ignored.	CODE_SMELL	MINOR	9

Collection.isEmpty() should be used to test for emptiness	Using Collection.size() to test for emptiness works, but using Collection.isEmpty() makes the code more readable and can be more performant. The time complexity of any isEmpty() method implementation should be O(1) whereas some implementations of size() can be O(n). Noncompliant Code Example if (myCollection.size() == 0) { // Noncompliant /* ... */ } Compliant Solution if (myCollection.isEmpty()) { /* ... */ }	CODE_SMELL	MINOR	4
Exception classes should be immutable	Exceptions are meant to represent the application's state at the point at which an error occurred. Making all fields in an Exception class final ensures that this state: Will be fully defined at the same time the Exception is instantiated. Won't be updated or corrupted by a questionable error handler. This will enable developers to quickly understand what went wrong. Noncompliant Code Example public class MyException extends Exception { private int status; // Noncompliant public MyException(String message) { super(message); } public int getStatus() { return status; } public void setStatus(int status) { this.status = status; } } Compliant Solution public class MyException extends Exception { private final int status; public MyException(String message, int status) { super(message); this.status = status; } public int getStatus() { return status; } }	CODE_SMELL	MINOR	3
Overriding methods should do more than simply call the same method in the super class	Overriding a method just to call the same method from the super class without performing any other actions is useless and misleading. The only time this is justified is in final overriding methods, where the effect is to lock in the parent class behavior. This rule ignores such overrides of equals, hashCode and toString. Noncompliant Code Example public void doSomething() { super.doSomething(); } @Override public boolean isLegal(Action action) { return super.isLegal(action); } Compliant Solution @Override public boolean isLegal(Action action) { // Compliant - not simply forwarding the call return super.isLegal(new Action(/* ... */)); } @Id @Override public int getId() { // Compliant - there is annotation different from @Override return super.getId(); }	CODE_SMELL	MINOR	1
Type parameter names should comply	Shared naming conventions make it possible for a team to collaborate efficiently. Following the	CODE_SMELL	MINOR	3

with a naming convention	established convention of single-letter type parameter names helps users and maintainers of your code quickly see the difference between a type parameter and a poorly named class. This rule check that all type parameter names match a provided regular expression. The following code snippets use the default regular expression. Noncompliant Code Example public class MyClass<TYPE> { // Noncompliant <TYPE> void method(TYPE t) { // Noncompliant } } Compliant Solution public class MyClass<T> { <T> void method(T t) { } }			
"switch" statements should have at least 3 "case" clauses	switch statements are useful when there are many different cases depending on the value of the same expression. For just one or two cases however, the code will be more readable with if statements. Noncompliant Code Example switch (variable) { case 0: doSomething(); break; default: doSomethingElse(); break; } Compliant Solution if (variable == 0) { doSomething(); } else { doSomethingElse(); }	CODE_SMELL	MINOR	4
Loops should not contain more than a single "break" or "continue" statement	Restricting the number of break and continue statements in a loop is done in the interest of good structured programming. Only one break or one continue statement is acceptable in a loop, since it facilitates optimal coding. If there is more than one, the code should be refactored to increase readability. Noncompliant Code Example for (int i = 1; i <= 10; i++) { // Noncompliant - 2 continue - one might be tempted to add some logic in between if (i % 2 == 0) { continue; } if (i % 3 == 0) { continue; } System.out.println("i = " + i); }	CODE_SMELL	MINOR	4
"public static" fields should be constant	There is no good reason to declare a field "public" and "static" without also declaring it "final". Most of the time this is a kludge to share a state among several objects. But with this approach, any object can do whatever it wants with the shared state, such as setting it to null. Noncompliant Code Example public class Greeter { public static Foo foo = new Foo(); ... } Compliant Solution public class Greeter { public static final Foo FOO = new Foo(); ... } See MITRE, CWE-500 - Public Static Field Not Marked Final CERT OBJ10-J. - Do not use public static nonfinal fields	CODE_SMELL	MINOR	3
Unused local variables	If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve	CODE_SMELL	MINOR	1

should be removed	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 Solution public int numberOfMinutes(int hours) { return hours * 60; }			
Local variables should not be declared and then immediately returned or thrown	Declaring a variable only to immediately return or throw it is a bad practice. Some developers argue that the practice improves code readability, because it enables them to explicitly name what is being returned. However, this variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned. Noncompliant Code Example public long computeDurationInMilliseconds() { long duration = (((hours * 60) + minutes) * 60 + seconds) * 1000 ; return duration; } public void doSomething() { RuntimeException myException = new RuntimeException(); throw myException; } Compliant Solution public long computeDurationInMilliseconds() { return (((hours * 60) + minutes) * 60 + seconds) * 1000 ; } public void doSomething() { throw new RuntimeException(); }	CODE_SMELL	MINOR	1
Lambdas should be replaced with method references	Method/constructor references are commonly agreed to be, most of the time, more compact and readable than using lambdas, and are therefore preferred. In some rare cases, when it is not clear from the context what kind of function is being described and reference would not increase the clarity, it might be fine to keep the lambda. Similarly, null checks can be replaced with references to the Objects::isNull and Objects::nonNull methods, casts can be replaced with SomeClass.class::cast and instanceof can be replaced with SomeClass.class::isInstance. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 8. Noncompliant Code Example class A { void process(List<A> list) { list.stream() .filter(a -> a instanceof B) .map(a -> (B) a) .map(b -> b.<String>getObject()) .forEach(b -> { System.out.println(b); }); } } class B extends A { <T> T getObject() { return null; } } Compliant Solution class A { void process(List<A> list) { list.stream() .filter(B.class::isInstance) .map(B.class::cast)	CODE_SMELL	MINOR	2

	.map(B::<String>getObject) .forEach(System.out::println); } } class B extends A { <T> T getObject() { return null; } }			
Multiple variables should not be declared on the same line	Declaring multiple variables on one line is difficult to read. Noncompliant Code Example class MyClass { private int a, b; public void method(){ int c; int d; } } Compliant Solution class MyClass { private int a; private int b; public void method(){ int c; int d; } } See CERT, DCL52-J. - Do not declare more than one variable per declaration CERT, DCL04-C. - Do not declare more than one variable per declaration	CODE_SMELL	MINOR	1
"@Deprecated" code should not be used	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 away from the aging, soon-to-be-retired technology. Noncompliant Code Example /** * @deprecated As of release 1.3, replaced by {@link #Fee} */ @Deprecated public 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 } } See MITRE, CWE-477 - Use of Obsolete Functions CERT, MET02-J. - Do not use deprecated or obsolete classes or methods	CODE_SMELL	MINOR	3
Classes should not be empty	There is no good excuse for an empty class. If it’s being used simply as a common extension point, it should be replaced with an interface. If it was stubbed in as a placeholder for future development it should be fleshed-out. In any other case, it should be eliminated. Noncompliant Code Example public class Nothing { // Noncompliant } Compliant Solution public interface Nothing { } Exceptions Empty classes can be used as marker types (for Spring for instance), therefore empty classes that are annotated will be ignored. @Configuration @EnableWebMvc public	CODE_SMELL	MINOR	1

	final class ApplicationConfiguration { }			
Subclasses that add fields should override "equals"	Extend a class that overrides equals and add fields without overriding equals in the subclass, and you run the risk of non-equivalent instances of your subclass being seen as equal, because only the superclass fields will be considered in the equality test. This rule looks for classes that do all of the following: extend classes that override equals. do not themselves override equals. add fields. Noncompliant Code Example public class Fruit { private Season ripe; public boolean equals(Object obj) { if (obj == this) { return true; } if (this.class != obj.class) { return false; } Fruit fobj = (Fruit) obj; if (ripe.equals(fobj.getRipe()) { return true; } return false; } } public class Raspberry extends Fruit { // Noncompliant; instances will use Fruit's equals method private Color ripeColor; } Compliant Solution public class Fruit { private Season ripe; public boolean equals(Object obj) { if (obj == this) { return true; } if (this.class != obj.class) { return false; } Fruit fobj = (Fruit) obj; if (ripe.equals(fobj.getRipe()) { return true; } return false; } } public class Raspberry extends Fruit { private Color ripeColor; public boolean equals(Object obj) { if (! super.equals(obj)) { return false; } Raspberry fobj = (Raspberry) obj; if (ripeColor.equals(fobj.getRipeColor()) { // added fields are tested return true; } return false; } }	CODE_SMELL	MINOR	1
The diamond operator ("<>") should be used	Java 7 introduced the diamond operator (<>) to reduce the verbosity of generics code. For instance, instead of having to declare a List's type in both its declaration and its constructor, you can now simplify the constructor declaration with <>, and the compiler will infer the type. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 7. Noncompliant Code Example List<String> strings = new ArrayList<String>(); // Noncompliant Map<String,List<Integer>> map = new HashMap<String,List<Integer>>(); // Noncompliant Compliant Solution List<String> strings = new ArrayList<>(); Map<String,List<Integer>> map = new HashMap<>();	CODE_SMELL	MINOR	5

Mutable fields should not be "public static"	There is no good reason to have a mutable object as the public (by default), static member of an interface. Such variables should be moved into classes and their visibility lowered. Similarly, mutable static members of classes and enumerations which are accessed directly, rather than through getters and setters, should be protected to the degree possible. That can be done by reducing visibility or making the field final if appropriate. Note that making a mutable field, such as an array, final will keep the variable from being reassigned, but doing so has no effect on the mutability of the internal state of the array (i.e. it doesn’t accomplish the goal). This rule raises issues for public static array, Collection, Date, and awt.Point members. Noncompliant Code Example public interface MyInterface { public static String [] strings; // Noncompliant } public class A { public static String [] strings1 = {"first","second"}; // Noncompliant public static String [] strings2 = {"first","second"}; // Noncompliant public static List<String> strings3 = new ArrayList<>(); // Noncompliant // ... } See MITRE, CWE-582 - Array Declared Public, Final, and Static MITRE, CWE-607 - Public Static Final Field References Mutable Object CERT, OBJ01-J. - Limit accessibility of fields CERT, OBJ13-J. - Ensure that references to mutable objects are not exposed	CODE_SMELL	MINOR	3
"catch" clauses should do more than rethrow	A catch clause that only rethrows the caught exception has the same effect as omitting the catch altogether and letting it bubble up automatically, but with more code and the additional detriment of leaving maintainers scratching their heads. Such clauses should either be eliminated or populated with the appropriate logic. Noncompliant Code Example public String readFile(File f) { StringBuilder sb = new StringBuilder(); try { FileReader fileReader = new FileReader(fileName); BufferedReader bufferedReader = new BufferedReader(fileReader); while((line = bufferedReader.readLine()) != null) { //... } catch (IOException e) { // Noncompliant throw e; } return sb.toString(); } Compliant Solution public String readFile(File f) { StringBuilder sb = new StringBuilder(); try { FileReader fileReader = new FileReader(fileName); BufferedReader bufferedReader = new BufferedReader(fileReader); while((line = bufferedReader.readLine()) != null) { //... } catch (IOException e) { logger.LogError(e); throw e; }	CODE_SMELL	MINOR	1

	return sb.toString(); } or public String readFile(File f) throws IOException { StringBuilder sb = new StringBuilder(); FileReader fileReader = new FileReader(fileName); BufferedReader bufferedReader = new BufferedReader(fileReader); while((line = bufferedReader.readLine()) != null) { //... return sb.toString(); }			
Jump statements should not be redundant	Jump statements such as return and continue let you change the default flow of program execution, but jump statements that direct the control flow to the original direction are just a waste of keystrokes. Noncompliant Code Example public void foo() { while (condition1) { if (condition2) { continue; // Noncompliant } else { doTheThing(); } } return; // Noncompliant; this is a void method } Compliant Solution public void foo() { while (condition1) { if (!condition2) { doTheThing(); } } }	CODE_SMELL	MINOR	1
Composed "@RequestMapping" variants should be preferred	Spring framework 4.3 introduced variants of the @RequestMapping annotation to better represent the semantics of the annotated methods. The use of @GetMapping, @PostMapping, @PutMapping, @PatchMapping and @DeleteMapping should be preferred to the use of the raw @RequestMapping(method = RequestMethod.XYZ). Noncompliant Code Example @RequestMapping(path = "/greeting", method = RequestMethod.GET) // Noncompliant public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) { ... } Compliant Solution @GetMapping(path = "/greeting") // Compliant public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) { ... }	CODE_SMELL	MINOR	13

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	0	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

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	0	0	0

SECURITY HOTSPOTS LIST