Code analysis

test-maker

Version 1.0

By: Administrator

Date: 2020-05-24

Introduction

This document contains results of the code analysis of test-maker

Configuration

- Quality Profiles
  - Names: Sonar way [Java];
  - Files: AXJEyxy6NN5stHLmh-tH.json;
- Quality Gate
  - Name: Sonar way
  - File: Sonar way.xml

Synthesis

Quality Gate	Reliability	Security	Maintainability	Coverage	Duplications
OK	D	B	A	21.6 %	1.7 %

Metrics

\	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Coverage	Comment density (%)	Duplication (%)
Min	0.0	0.0	3.0	0.0	0.0	0.0
Max	2549.0	1324.0	13283.0	100.0	66.7	62.9

Volume

Language	Number
Java	13283
Total	13283

Issues count by severity and types

Type	Severity	Number
VULNERABILITY	BLOCKER	0
VULNERABILITY	CRITICAL	0
VULNERABILITY	MAJOR	0
VULNERABILITY	MINOR	11
VULNERABILITY	INFO	0
BUG	BLOCKER	0
BUG	CRITICAL	1
BUG	MAJOR	21
BUG	MINOR	7
BUG	INFO	0
CODE_SMELL	BLOCKER	5
CODE_SMELL	CRITICAL	51
CODE_SMELL	MAJOR	173
CODE_SMELL	MINOR	278
CODE_SMELL	INFO	13
SECURITY_HOTSPOT	BLOCKER	0
SECURITY_HOTSPOT	CRITICAL	0
SECURITY_HOTSPOT	MAJOR	0
SECURITY_HOTSPOT	MINOR	0
SECURITY_HOTSPOT	INFO	0

Issues

Name	Description	Type	Severity	Number

"Random" objects should be reused	Creating a new Random object each time a random value is needed is inefficient and may produce numbers which are not random depending on the JDK. For better efficiency and randomness, create a single Random, then store, and reuse it.
	The Random() constructor tries to set the seed with a distinct value every time. However there is no guarantee that the seed will be random or even uniformly distributed. Some JDK will use the current time as seed, which makes the generated numbers not random at all.
	This rule finds cases where a new Random is created each time a method is invoked and assigned to a local random variable.
	Noncompliant Code Example
	
	public void doSomethingCommon() {
	Random rand = new Random(); // Noncompliant; new instance created with each invocation
	int rValue = rand.nextInt();
	//...
	
	Compliant Solution
	
	private Random rand = SecureRandom.getInstanceStrong(); // SecureRandom is preferred to Random
	
	public void doSomethingCommon() {
	int rValue = this.rand.nextInt();
	//...
	
	Exceptions
	A class which uses a Random in its constructor or in a static main function and nowhere else will be ignored by this
	rule.
	See
	
	OWASP Top 10 2017 Category A6 - Security Misconfiguration
		BUG	CRITICAL	1

Identical expressions should not be used on both sides of a binary operator	Using the same value on either side of a binary operator is almost always a mistake. In the case of logical operators, it is either a copy/paste
	error and therefore a bug, or it is simply wasted code, and should be simplified. In the case of bitwise operators and most binary mathematical
	operators, having the same value on both sides of an operator yields predictable results, and should be simplified.
	Noncompliant Code Example
	
	if (a == a) { // always true
	doZ();
	}
	if (a != a) { // always false
	doY();
	}
	if (a == b && a == b) { // if the first one is true, the second one is too
	doX();
	}
	if (a == b &#124&#124 a == b) { // if the first one is true, the second one is too
	doW();
	}
	
	int j = 5 / 5; //always 1
	int k = 5 - 5; //always 0
	
	c.equals(c); //always true
	
	Exceptions
		BUG	MAJOR	1

	This rule ignores *, +, and =.
	The specific case of testing a floating point value against itself is a valid test for NaN and is therefore ignored.
	Similarly, left-shifting 1 onto 1 is common in the construction of bit masks, and is ignored.
	
	float f;
	if(f != f) { //test for NaN value
	System.out.println("f is NaN");
	}
	
	int i = 1 << 1; // Compliant
	int j = a << a; // Noncompliant
	
	See
	
	CERT, MSC12-C. - Detect and remove code that has no effect or is never
	executed
	S1656 - Implements a check on =.			

Null pointers should not be dereferenced	A reference to null should never be dereferenced/accessed. Doing so will cause a NullPointerException to be thrown. At
	best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or
	it could allow an attacker to bypass security measures.
	Note that when they are present, this rule takes advantage of @CheckForNull and @Nonnull annotations defined in JSR-305 to understand which values are and are not nullable except when @Nonnull is used
	on the parameter to equals, which by contract should always work with null.
	Noncompliant Code Example
	
	@CheckForNull
	String getName(){...}
	
	public boolean isNameEmpty() {
	return getName().length() == 0; // Noncompliant; the result of getName() could be null, but isn't null-checked
	}
	
	Connection conn = null;
	Statement stmt = null;
	try{
	conn = DriverManager.getConnection(DB_URL,USER,PASS);
	stmt = conn.createStatement();
	// ...
	
	}catch(Exception e){
	e.printStackTrace();
	}finally{
	stmt.close(); // Noncompliant; stmt could be null if an exception was thrown in the try{} block
	conn.close(); // Noncompliant; conn could be null if an exception was thrown
	}
	
	private void merge(@Nonnull Color firstColor, @Nonnull Color secondColor){...}
	
	public void append(@CheckForNull Color color) {
	merge(currentColor, color); // Noncompliant; color should be null-
		BUG	MAJOR	15

	checked because merge(...) doesn't accept nullable parameters
	}
	void paint(Color color) {
	if(color == null) {
	System.out.println("Unable to apply color " + color.toString()); // Noncompliant; NullPointerException will be thrown
	return;
	}
	...
	}
	See
	MITRE, CWE-476 - NULL Pointer Dereference
	CERT, EXP34-C. - Do not dereference null pointers
	CERT, EXP01-J. - Do not use a null in a case where an object is required			

Conditionally executed code should be reachable	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 Example
	a = false;
	if (a) { // Noncompliant
	doSomething(); // never executed
	}
	if (!a &#124&#124 b) { // Noncompliant; "!a" is always "true", "b" is never evaluated
	doSomething();
	} else {
	doSomethingElse(); // never executed
	}
	Exceptions
	This rule will not raise an issue in either of these cases:
	When the condition is a single final boolean
	final boolean debug = false;
	//...
	if (debug) {
	// Print something
	}
	When the condition is literally true or false.
	if (true) {
	// do something
	}
	In these cases it is obvious the code is as intended.
	See
	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
		BUG	MAJOR	3

	Not all classes in the standard Java library were written to be thread-			

Non-thread-safe fields should not be static	safe. Using them in a multi-threaded manner is highly likely to cause data
	problems or exceptions at runtime.
	This rule raises an issue when an instance of Calendar, DateFormat, javax.xml.xpath.XPath, or
	javax.xml.validation.SchemaFactory is marked static.
	Noncompliant Code Example
	
	public class MyClass {
	private static SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss"); // Noncompliant
	private static Calendar calendar = Calendar.getInstance(); // Noncompliant
	
	Compliant Solution
	
	public class MyClass {
	private SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss");
	private Calendar calendar = Calendar.getInstance();
		BUG	MAJOR	2

Math operands should be cast before assignment	When arithmetic is performed on integers, the result will always be an integer. You can assign that result to a long,
	double, or float with automatic type conversion, but having started as an int or long, the result
	will likely not be what you expect.
	For instance, if the result of int division is assigned to a floating-point variable, precision will have been lost before the
	assignment. Likewise, if the result of multiplication is assigned to a long, it may have already overflowed before the assignment.
	In either case, the result will not be what was expected. Instead, at least one operand should be cast or promoted to the final type before the operation takes place.
	Noncompliant Code Example
	
	float twoThirds = 2/3; // Noncompliant; int division. Yields 0.0
	long millisInYear = 1_000*3_600*24*365; // Noncompliant; int multiplication. Yields 1471228928 <br /> long bigNum = Integer.MAX_VALUE + 2; // Noncompliant. Yields -2147483647 <br /> long bigNegNum = Integer.MIN_VALUE-1; //Noncompliant, gives a positive result instead of a negative one. <br /> Date myDate = new Date(seconds * 1_000); //Noncompliant, won't produce the expected result if seconds > 2_147_483
	...
	public long compute(int factor){
	return factor * 10_000; //Noncompliant, won't produce the expected result if factor > 214_748
	}
	
	public float compute2(long factor){
	return factor / 123; //Noncompliant, will be rounded to closest long integer
	}
	
	Compliant Solution
	
	float twoThirds = 2f/3; // 2 promoted to float. Yields 0.6666667
	long millisInYear = 1_000L*3_600*24*365; // 1000 promoted to long. Yields 31_536_000_000 <br /> long bigNum = Integer.MAX_VALUE + 2L; // 2 promoted to long. Yields 2_147_483_649 <br /> long bigNegNum = Integer.MIN_VALUE-1L; // Yields -2_147_483_649 <br /> Date myDate = new Date(seconds * 1_000L);
	...
	public long compute(int factor){
	return factor * 10_000L;
	}
		BUG	MINOR	6

	public float compute2(long factor){
	return factor / 123f;
	}
	
	or
	
	float twoThirds = (float)2/3; // 2 cast to float
	long millisInYear = (long)1_000*3_600*24*365; // 1_000 cast to long
	long bigNum = (long)Integer.MAX_VALUE + 2;
	long bigNegNum = (long)Integer.MIN_VALUE-1;
	Date myDate = new Date((long)seconds * 1_000);
	...
	public long compute(long factor){
	return factor * 10_000;
	}
	
	public float compute2(float factor){
	return factor / 123;
	}
	
	See
	
	MITRE, CWE-190 - Integer Overflow or Wraparound
	CERT, NUM50-J. - Convert integers to floating point for floating-point operations
	CERT, INT18-C. - Evaluate integer expressions in a larger size before comparing or assigning to that size
	SANS Top 25 - Risky Resource Management			

"@NonNull" values should not be set to null	Fields, parameters and return values marked @NotNull, @NonNull, or @Nonnull are assumed to have non-null
	values and are not typically null-checked before use. Therefore setting one of these values to null, or failing to set such a class field
	in a constructor, could cause NullPointerExceptions at runtime.
	Noncompliant Code Example
	
	public class MainClass {
	
	@Nonnull
	private String primary;
	private String secondary;
	
	public MainClass(String color) {
	if (color != null) {
	secondary = null;
	}
	primary = color; // Noncompliant; "primary" is Nonnull but could be set to null here
	}
	
	public MainClass() { // Noncompliant; "primary" Nonnull" but is not initialized
	}
	
	@Nonnull
	public String indirectMix() {
	String mix = null;
	return mix; // Noncompliant; return value is Nonnull, but null is returned.}}
	}
	
	See
	
	MITRE CWE-476 - NULL Pointer Dereference
	CERT, EXP01-J. - Do not use a null in a case where an object is required
		BUG	MINOR	1

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	2

Methods returns should not be invariant	When a method is designed to return an invariant value, it may be poor
	design, but it shouldn't adversely affect the outcome of your program.
	However, when it happens on all paths through the logic, it is surely a
	bug.
	This rule raises an issue when a method contains several return
	statements that all return the same value.
	Noncompliant Code Example
	
	int foo(int a) {
	int b = 12;
	if (a == 1) {
	return b;
	}
	return b; // Noncompliant
	}
		CODE_SMELL	BLOCKER	3

	Shared coding conventions allow teams to collaborate efficiently. This rule checks that all constant names match a provided regular expression.			

Constant names should comply with a naming convention	Noncompliant Code Example
	With the default regular expression ^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$:
	
	public class MyClass {
	public static final int first = 1;
	}
	
	public enum MyEnum {
	first;
	}
	
	Compliant Solution
	
	public class MyClass {
	public static final int FIRST = 1;
	}
	
	public enum MyEnum {
	FIRST;
	}
		CODE_SMELL	CRITICAL	3

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 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
	}
	}
		CODE_SMELL	CRITICAL	23

	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			

String literals should not be duplicated	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	14

Constants should not be defined in interfaces	According to Joshua Bloch, author of "Effective Java":
	
	The constant interface pattern is a poor use of interfaces.
	That a class uses some constants internally is an implementation detail.
	Implementing a constant interface causes this implementation detail to leak into the class's exported API. It is of no consequence to the users of a class that the class implements a constant interface. In fact, it may even confuse them. Worse, it represents a commitment: if in a future release the class is modified so that it no longer needs to use the constants, it still must implement the interface to ensure binary compatibility.
	If a nonfinal class implements a constant interface,
	all of its subclasses will have their namespaces polluted by the constants in the interface.
	
	Noncompliant Code Example
	
	interface Status { // Noncompliant
	int OPEN = 1;
	int CLOSED = 2;
	}
	
	Compliant Solution
	
	public enum Status { // Compliant
	OPEN,
	CLOSED;
	}
	
	or
	
	public final class Status { // Compliant
	public static final int OPEN = 1;
	public static final int CLOSED = 2;
	}
		CODE_SMELL	CRITICAL	1

	Fields in a Serializable class must themselves be either Serializable or transient even if the class is			

Fields in a "Serializable" class should either be transient or serializable	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	3

"static" base class members	In the interest of code clarity, static members of a base class should never be accessed using a derived type's name.
	Doing so is confusing and could create the illusion that two different static members exist.
	Noncompliant Code Example
	
	class Parent {
	public static int counter;
	}
	
	class Child extends Parent {
	public Child() {
	Child.counter++; // Noncompliant
	}			

should not be accessed via derived types	}
	
	Compliant Solution
	
	class Parent {
	public static int counter;
	}
	
	class Child extends Parent {
	public Child() {
	Parent.counter++;
	}
	}
		CODE_SMELL	CRITICAL	1

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	6

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	13

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	17

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
	
	CERT, ERR02-J. - Prevent exceptions while logging data
		CODE_SMELL	MAJOR	2

	Merging collapsible if statements increases the code's readability. Noncompliant Code Example			

Collapsible "if" statements should be merged	if (file != null) {
	if (file.isFile() &#124&#124 file.isDirectory()) {
	/* ... /
	}
	}
	
	Compliant Solution
	
	if (file != null && isFileOrDirectory(file)) {
	/ ... */
	}
	
	private static boolean isFileOrDirectory(File file) {
	return file.isFile() &#124&#124 file.isDirectory();
	}
		CODE_SMELL	MAJOR	11

Unused "private" fields should be removed	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 Example
	
	public class MyClass {
	private int foo = 42;
	
	public int compute(int a) {
	return a * 42;
	}
	
	}
	
	Compliant Solution
	
	public class MyClass {
	public int compute(int a) {
	return a * 42;
	}
	}
	
	Exceptions
	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;
	}
	
	Moreover, this rule doesn't raise any issue on annotated fields.
		CODE_SMELL	MAJOR	9

Nested blocks of code should not be left empty	Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed. Noncompliant Code Example for (int i = 0; i < 42; i++){} // Empty on purpose or missing piece of code ?	CODE_SMELL	MAJOR	1

	Exceptions When a block contains a comment, this block is not considered to be empty unless it is a synchronized block. synchronized blocks are still considered empty even with comments because they can still affect program flow.			
Inheritance tree of classes should not be too deep	Inheritance is certainly one of the most valuable concepts in object-oriented programming. It's a way to compartmentalize and reuse code by creating collections of attributes and behaviors called classes which can be based on previously created classes. But abusing this concept by creating a deep inheritance tree can lead to very complex and unmaintainable source code. Most of the time a too deep inheritance tree is due to bad object oriented design which has led to systematically use 'inheritance' when for instance 'composition' would suit better. This rule raises an issue when the inheritance tree, starting from Object has a greater depth than is allowed.	CODE_SMELL	MAJOR	1

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
	
	CERT, DCL01-C. - Do not reuse
	variable names in subscopes
	CERT, DCL51-J. - Do
	not shadow or obscure identifiers in subscopes
		CODE_SMELL	MAJOR	4

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) {
		CODE_SMELL	MAJOR	8

	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.			

Labels should not be used	Labels are not commonly used in Java, and many developers do not understand how they work. Moreover, their usage makes the control flow harder to
	follow, which reduces the code's readability.
	Noncompliant Code Example
	
	int matrix[][] = {
	{1, 2, 3},
	{4, 5, 6},
	{7, 8, 9}
	};
	
	outer: for (int row = 0; row < matrix.length; row++) { // Non-Compliant
	for (int col = 0; col < matrix[row].length; col++) {
	if (col == row) {
	continue outer;
	}
	System.out.println(matrix[row][col]); // Prints the elements under the diagonal, i.e. 4, 7 and 8
	}
	}
	
	Compliant Solution
	
	for (int row = 1; row < matrix.length; row++) { // Compliant
	for (int col = 0; col < row; col++) {
	System.out.println(matrix[row][col]); // Also prints 4, 7 and 8
	}
	}
		CODE_SMELL	MAJOR	2

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
		CODE_SMELL	MAJOR	20

	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			

Assignments should not be made from within sub-expressions	Assignments within sub-expressions are hard to spot and therefore make the code less readable. Ideally, sub-expressions should not have side-effects.
	Noncompliant Code Example
	
	if ((str = cont.substring(pos1, pos2)).isEmpty()) { // Noncompliant
	//...
	
	Compliant Solution
	
	str = cont.substring(pos1, pos2);
	if (str.isEmpty()) {
	//...
	
	Exceptions
	Assignments in while statement conditions, and assignments enclosed in relational expressions are ignored.
	
	BufferedReader br = new BufferedReader(/* ... */);
	String line;
	while ((line = br.readLine()) != null) {...}
	
	Chained assignments, including compound assignments, are ignored.
	
	int i = j = 0;
	int k = (j += 1);
	result = (bresult = new byte[len]);
	
	See
	
	MITRE, CWE-481 - Assigning instead of Comparing
	CERT, EXP45-C. - Do not perform assignments in selection statements
	CERT, EXP51-J. - Do not perform assignments in conditional expressions
		CODE_SMELL	MAJOR	1

Track uses of "FIXME" tags	FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with 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
	
	int divide(int numerator, int denominator) {
	return numerator / denominator; // FIXME denominator value might be 0
	}
	
	See
	
	MITRE, CWE-546 - Suspicious Comment
		CODE_SMELL	MAJOR	1

	private methods that are never executed are dead code: unnecessary, inoperative code that should be removed. Cleaning out dead code			

Unused "private" methods should be removed	decreases the size of the maintained codebase, making it easier to understand the program and preventing bugs from being introduced.
	Note that this rule does not take reflection into account, which means that issues will be raised on private methods that are only
	accessed using the reflection API.
	Noncompliant Code Example
	
	public class Foo implements Serializable
	{
	private Foo(){} //Compliant, private empty constructor intentionally used to prevent any direct instantiation of a class.
	public static void doSomething(){
	Foo foo = new Foo();
	...
	}
	private void unusedPrivateMethod(){...}
	private void writeObject(ObjectOutputStream s){...} //Compliant, relates to the java serialization mechanism
	private void readObject(ObjectInputStream in){...} //Compliant, relates to the java serialization mechanism
	}
	
	Compliant Solution
	
	public class Foo implements Serializable
	{
	private Foo(){} //Compliant, private empty constructor intentionally used to prevent any direct instantiation of a class.
	public static void doSomething(){
	Foo foo = new Foo();
	...
	}
	
	private void writeObject(ObjectOutputStream s){...} //Compliant, relates to the java serialization mechanism
	
	private void readObject(ObjectInputStream in){...} //Compliant, relates to the java serialization mechanism
	}
	
	Exceptions
	This rule doesn't raise any issue on annotated methods.
		CODE_SMELL	MAJOR	4

Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	Early classes of the Java API, such as Vector, Hashtable and StringBuffer, were synchronized to make them thread-safe. Unfortunately, synchronization has a big negative impact on performance, even when using these collections from a single thread. It is better to use their new unsynchronized replacements: ArrayList or LinkedList instead of Vector Deque instead of Stack HashMap instead of Hashtable StringBuilder instead of StringBuffer Noncompliant Code Example Vector cats = new Vector(); Compliant Solution ArrayList cats = new ArrayList(); Exceptions Use of those synchronized classes is ignored in the signatures of overriding methods. @Override public Vector getCats() {...}	CODE_SMELL	MAJOR	1

Empty arrays and collections should be returned instead of null	Returning null instead of an actual array or collection 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> getResults() {
	return null; // Noncompliant
	}
	
	public static Result[] getResults() {
	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) {
	/* ... /
	}
	}
	}
	
	Compliant Solution
	
	public static List<Result> getResults() {
	return Collections.emptyList(); // Compliant
	}
	
	public static Result[] getResults() {
	return new Result[0];
	}
	
	public static void main(String[] args) {
	for (Result result: getResults()) {
	/ ... */
	}
	}
	
	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	3

	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 @javax.enterprise.event.Observes			

Unused method parameters should be removed	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
		CODE_SMELL	MAJOR	2

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	71

Anonymous inner classes containing only one method should become lambdas	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 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) {
		CODE_SMELL	MAJOR	2

	return myString.isEmpty();
	}
	}
	
	Compliant Solution
	
	myCollection.stream().map(input -> new StringBuilder(input).reverse().toString());
	
	Predicate<String> isEmpty = myString -> myString.isEmpty();			

Static fields should not be updated in constructors	Assigning a value to a static field in a constructor could cause unreliable behavior at runtime since it will change the value for all
	instances of the class.
	Instead remove the field's static modifier, or initialize it statically.
	Noncompliant Code Example
	
	public class Person {
	static Date dateOfBirth;
	static int expectedFingers;
	
	public Person(date birthday) {
	dateOfBirth = birthday; // Noncompliant; now everyone has this birthday
	expectedFingers = 10; // Noncompliant
	}
	}
	
	Compliant Solution
	
	public class Person {
	Date dateOfBirth;
	static int expectedFingers = 10;
	
	public Person(date birthday) {
	dateOfBirth = birthday;
	}
	}
		CODE_SMELL	MAJOR	1

String function use should be optimized for single characters	An indexOf or lastIndexOf call with a single letter String can be made more performant by switching to a
	call with a char argument.
	Noncompliant Code Example
	
	String myStr = "Hello World";
	// ...
	int pos = myStr.indexOf("W"); // Noncompliant
	// ...
	int otherPos = myStr.lastIndexOf("r"); // Noncompliant
	// ...
	
	Compliant Solution
	
	String myStr = "Hello World";
	// ...
	int pos = myStr.indexOf('W');
	// ...
	int otherPos = myStr.lastIndexOf('r');
	// ...
		CODE_SMELL	MAJOR	1

	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			

Printf-style format strings should be used correctly	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$dandthen$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
		CODE_SMELL	MAJOR	2

Raw types should not be used	Generic types shouldn't be used raw (without type parameters) in variable declarations or return values. Doing so bypasses generic type checking,
	and defers the catch of unsafe code to runtime.
	Noncompliant Code Example
	
	List myList; // Noncompliant
	Set mySet; // Noncompliant
	
	Compliant Solution
	
	List<String> myList;
	Set<? extends Number> mySet;
		CODE_SMELL	MAJOR	7

"java.nio.Files#delete" should be preferred	When java.io.File#delete fails, this boolean method simply returns false with no indication of the cause. On
	the other hand, when java.nio.file.Files#delete fails, this void method returns one of a series of exception types to better
	indicate the cause of the failure. And since more information is generally better in a debugging situation, java.nio.file.Files#delete is the preferred option.
	Noncompliant Code Example
	
	public void cleanUp(Path path) {
	File file = new File(path);
	if (!file.delete()) { // Noncompliant
	//...
	}
	}
	
	Compliant Solution
		CODE_SMELL	MAJOR	1

	public void cleanUp(Path path) throws NoSuchFileException, DirectoryNotEmptyException, IOException {
	Files.delete(path);
	}			

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

Method names should comply with a naming convention	Shared naming conventions allow teams to collaborate efficiently. This rule checks that all method names match a provided regular expression. Noncompliant Code Example With default provided regular expression ^[a-z][a-zA-Z0-9]*$: public int DoSomething(){...} Compliant Solution public int doSomething(){...} Exceptions Overriding methods are excluded. @Override public int Do_Something(){...}	CODE_SMELL	MINOR	7

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 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
	...
	}
		CODE_SMELL	MINOR	3

	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	6

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
		CODE_SMELL	MINOR	141

	Imports for types mentioned in comments, such as Javadocs, are ignored.			

"throws" declarations should not be superfluous	An exception in a throws declaration in Java is superfluous if it is:
	
	listed multiple times
	a subclass of another listed exception
	a RuntimeException, or one of its descendants
	completely unnecessary because the declared exception type cannot actually be thrown
	
	Noncompliant Code Example
	
	void foo() throws MyException, MyException {} // Noncompliant; should be listed once
	void bar() throws Throwable, Exception {} // Noncompliant; Exception is a subclass of Throwable
	void baz() throws RuntimeException {} // Noncompliant; RuntimeException can always be thrown
	
	Compliant Solution
	
	void foo() throws MyException {}
	void bar() throws Throwable {}
	void baz() {}
	
	Exceptions
	The rule will not raise any issue for exceptions that cannot be thrown from the method body:
	
	in overriding and implementation methods
	in interface default methods
	in non-private methods that only throw, have empty bodies, or a single return statement .
	in overridable methods (non-final, or not member of a final class, non-static, non-private), if the exception is documented with a proper javadoc.
	
	class A extends B {
	@Override
	void doSomething() throws IOException {
	compute(a);
	}
	
	public void foo() throws IOException {}
	
	protected void bar() throws IOException {
	throw new UnsupportedOperationException("This method should be implemented in subclasses");
	}
	
	Object foobar(String s) throws IOException {
	return null;
	}
	
	/**
	* @throws IOException Overriding classes may throw this exception if they print values into a file
	*/
	protected void print() throws IOException { // no issue, method is overridable and the exception has proper javadoc
	System.out.println("foo");
	}
	}
		CODE_SMELL	MINOR	2

	Using Collection.size() to test for emptiness works, but using Collection.isEmpty() makes the code more readable and can			

Collection.isEmpty() should be used to test for emptiness	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

Field names should comply with a naming convention	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.
	Noncompliant Code Example
	With the default regular expression ^[a-z][a-zA-Z0-9]*$:
	
	class MyClass {
	private int my_field;
	}
	
	Compliant Solution
	
	class MyClass {
	private int myField;
	}
		CODE_SMELL	MINOR	5

Local variable and method parameter names should comply with a naming convention	Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when a local variable or function parameter name does
	not match the provided regular expression.
	Noncompliant Code Example
	With the default regular expression ^[a-z][a-zA-Z0-9]*$:
	
	public void doSomething(int my_param) {
	int LOCAL;
	...
	}
	
	Compliant Solution
	
	public void doSomething(int myParam) {
	int local;
	...
	}
	
	Exceptions
	Loop counters are ignored by this rule.
	
	for (int i_1 = 0; i_1 < limit; i_1++) { // Compliant
	// ...
	}
	
	as well as one-character catch variables:
	
	try {
	//...
	} catch (Exception e) { // Compliant
	}
		CODE_SMELL	MINOR	13

	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			

Overriding methods should do more than simply call the same method in the super class	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 with a naming convention	Shared naming conventions make it possible for a team to collaborate efficiently. Following the 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) {
	}
	}
		CODE_SMELL	MINOR	2

Array designators "[]" should be on the type, not the variable	Array designators should always be located on the type for better code readability. Otherwise, developers must look both at the type and the variable name to know whether or not a variable is an array.
	Noncompliant Code Example
	
	int matrix[][]; // Noncompliant
	int[] matrix[]; // Noncompliant
	
	Compliant Solution
	
	int[][] matrix; // Compliant
		CODE_SMELL	MINOR	1

Package names should comply with a naming convention	Shared coding conventions allow teams to collaborate efficiently. This rule checks that all package names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z]+(.[a-z][a-z0-9_])$: package org.exAmple; // Noncompliant Compliant Solution	CODE_SMELL	MINOR	30

	package org.example;			

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.
	
	One break and 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	1

Private fields only used as local variables in methods should become local variables	When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding.
	Noncompliant Code Example
	
	public class Foo {
	private int a;
	private int b;
	
	public void doSomething(int y) {
	a = y + 5;
	...
	if(a == 0) {
	...
	}
	...
	}
	
	public void doSomethingElse(int y) {
	b = y + 3;
	...
	}
	}
	
	Compliant Solution
	
	public class Foo {
	
	public void doSomething(int y) {
	int a = y + 5;
	...
	if(a == 0) {
	...
	}
	}
	
	public void doSomethingElse(int y) {
	int b = y + 3;
	...
	}
	}
	
	Exceptions
		CODE_SMELL	MINOR	2

	This rule doesn't raise any issue on annotated field.			

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	2

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

	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			

Classes should not be empty	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 final class ApplicationConfiguration {
	}
		CODE_SMELL	MINOR	1

Methods of "Random" that return floating point values should not be used in random integer generation	There is no need to multiply the output of Random's nextDouble method to get a random integer. Use the nextInt method instead.
	This rule raises an issue when the return value of any of Random's methods that return a floating point value is converted to an integer.
	Noncompliant Code Example
	Random r = new Random();
	int rand = (int)r.nextDouble() * 50; // Noncompliant way to get a pseudo-random value between 0 and 50
	int rand2 = (int)r.nextFloat(); // Noncompliant; will always be 0;
	Compliant Solution
	Random r = new Random();
	int rand = r.nextInt(50); // returns pseudo-random value between 0 and 50
		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	9

	According to the docs: Nested enum types are implicitly static. So there's no need to declare them static explicitly. Noncompliant Code Example public class Flower {			

Nested "enum"s should not be declared static	static enum Color { // Noncompliant; static is redundant here
	RED, YELLOW, BLUE, ORANGE
	}
	
	// ...
	}
	
	Compliant Solution
	
	public class Flower {
	enum Color { // Compliant
	RED, YELLOW, BLUE, ORANGE
	}
	
	// ...
	}
		CODE_SMELL	MINOR	1

Arrays should not be copied using loops	Using a loop to copy an array or a subset of an array is simply wasted code when there are built-in functions to do it for you. Instead, use Arrays.copyOf to copy an entire array into another array, use System.arraycopy to copy only a subset of an array into
	another array, and use Arrays.asList to feed the constructor of a new list with an array.
	Note that Arrays.asList simply puts a Collections wrapper around the original array, so further steps are required if a
	non-fixed-size List is desired.
	Noncompliant Code Example
	
	public void makeCopies(String[] source) {
	
	this.array = new String[source.length];
	this.list = new ArrayList(source.length);
	
	for (int i = 0; i < source.length; i++) {
	this.array[i] = source[i]; // Noncompliant
	}
	
	for (String s : source) {
	this.list.add(s); // Noncompliant
	}
	}
	
	Compliant Solution
	
	public void makeCopies(String[] source) {
	this.array = Arrays.copyOf(source, source.length);
	Collections.addAll(this.list, source);
	}
	
	Exceptions
	Rule detects only the most idiomatic patterns, it will not consider loops with non-trivial control flow. For example, array elements that are copied conditionally are ignored.
	
	public int[] getCopy(int[] source) {
	int[] dest = new int[source.length];
	for (int i = 0; i < source.length; i++) {
	if (source[i] > 10) {
	dest[i] = source[i]; // Compliant
	}
	}
	return dest;
	}
		CODE_SMELL	MINOR	1

	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			

Jump statements should not be redundant	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	37

Arrays should not be created for varargs parameters	There's no point in creating an array solely for the purpose of passing it as a varargs (...) argument; varargs is an array.
	Simply pass the elements directly. They will be consolidated into an array automatically. Incidentally passing an array where Object ...
	is expected makes the intent ambiguous: Is the array supposed to be one object or a collection of objects?
	Noncompliant Code Example
	
	public void callTheThing() {
	//...
	doTheThing(new String[] { "s1", "s2"}); // Noncompliant: unnecessary
	doTheThing(new String[12]); // Compliant
	doTheOtherThing(new String[8]); // Noncompliant: ambiguous
	// ...
	}
	
	public void doTheThing (String ... args) {
	// ...
	}
	
	public void doTheOtherThing(Object ... args) {
	// ...
	}
	
	Compliant Solution
	
	public void callTheThing() {
	//...
	doTheThing("s1", "s2");
	doTheThing(new String[12]);
	doTheOtherThing((Object[]) new String[8]);
	// ...
	}
	
	public void doTheThing (String ... args) {
	// ...
	}
	
	public void doTheOtherThing(Object ... args) {
	// ...
	}
		CODE_SMELL	MINOR	3

	When boxed type java.lang.Boolean is used as an expression it will			

Boxed "Boolean" should be avoided in boolean expressions	throw NullPointerException if the value is
	null as defined in Java Language Specification §5.1.8
	Unboxing Conversion.
	It is safer to avoid such conversion altogether and handle the null value explicitly.
	Noncompliant Code Example
	
	Boolean b = getBoolean();
	if (b) { // Noncompliant, it will throw NPE when b == null
	foo();
	} else {
	bar();
	}
	
	Compliant Solution
	
	Boolean b = getBoolean();
	if (Boolean.TRUE.equals(b)) {
	foo();
	} else {
	bar(); // will be invoked for both b == false and b == null
	}
	
	See
	* Java Language Specification §5.1.8 Unboxing Conversion
		CODE_SMELL	MINOR	4

Throwable.printStackTrace(...) should not be called	Throwable.printStackTrace(...) prints a Throwable and its stack trace to some stream. By default that stream
	System.Err, which could inadvertently expose sensitive information.
	Loggers should be used instead to print Throwables, as they have many advantages:
	
	Users are able to easily retrieve the logs.
	The format of log messages is uniform and allow users to browse the logs easily.
	
	This rule raises an issue when printStackTrace is used without
	arguments, i.e. when the stack trace is printed to the default
	stream.
	Noncompliant Code Example
	
	try {
	/* ... /
	} catch(Exception e) {
	e.printStackTrace(); // Noncompliant
	}
	
	Compliant Solution
	
	try {
	*/ ... */*
	} catch(Exception e) {
	LOGGER.log("context", e);
	}
	
	See
	
	OWASP Top 10 2017 Category A3 - Sensitive Data Exposure
	
	MITRE, CWE-489 - Leftover Debug Code
		VULNERABILITY	MINOR	11