Junta de Andalucía

Normativa de Análisis del Código con SONAR

Pruebas Estáticas

OAC – Oficina de Calidad

Versión: 01.04

Fecha: 23/12/2024

HOJA DE CONTROL

Documento	Normativa de Análisis del Código con SONAR / Pruebas Estáticas	**Versión**	01.04
Proyecto	OAC - Oficina de Calidad		
Elaborado por	Luis Albea Carlini (SOPRA)	**Fecha**	10/01/2017
Validado por		**Fecha**	
Aprobado por		**Fecha**	
Fichero	Normativa de Análisis del Código con SONAR_v01.04.odt		
Plantilla	OAC-Formato_Vertical-PLT	**Versión**	01.02

Control de Modificaciones

Versión	Fecha	Autor	Descripción del Cambio
01.00	10/01/2017	Luis Albea Carlini (SOPRA)	Versión inicial
01.01	16/01/2017	Luis Albea Carlini (SOPRA)	Se elimina la métrica de % de documentación de la API por dejar de generarla SonarQube a partir de la versión del plugin de java 4.3 Se modifica el umbral de calidad de CMAOT QG_0200 a CMAOT QG_2.1.0 Se indican los criterios de aceptación o rechazo de la entrega
01.03	01/03/3017	Luis Albea Carlini (SOPRA)	Revisión sintáctica del documento
01.03	10/01/2022	Mejora Continua	Actualización de logos.
01.03	21/11/2024	Oficina de Calidad	Actualización de información correspondiente a la nueva estrategia Sonar establecida y aprobada por Consejería
01.04	23/12/2024	Oficina de Calidad	Aclaración adaptación métricas ADA

Índice

1. Introducción

1.1. Objeto y Alcance

El objeto del presente documento es establecer la normativa para la realización de las verificaciones de calidad del código Java haciendo uso de SonarQube como herramienta específica para el análisis estático del código, con el objetivo de mejorar la calidad del mismo y controlar su evolución y mantenibilidad a lo largo del tiempo.

La versión de SONAR, sobre la que se basa el presente documento, es SonarQube 9.9.3.

1.2. Referencias

Documento	Fichero
Verificación entrega de software. Sonar	http://madeja.i-administracion.junta-andalucia.es/servicios/madeja/contenido/recurso/372
Libro Blanco de Desarrollo	https://www.cma.junta-andalucia.es/medioambiente/portal/web/portal-conocimiento/reglas-generales
Página Oficial de SonarQube	http://www.sonarqube.org/
Plantilla Informe de Análisis estático de código con Sonar	OAC-ICS-Informe_Sonar-PLT
Guía de uso de SonarQube	OAC-GEN-Análisis de Código con SonarQube

2. Perfiles SonarQube para Java

En SonarQube, se define perfil al conjunto de reglas que se aplican al código de un proyecto para realizar el examen estático de su calidad. Al incumplimiento de cada una de las reglas se le asigna un valor de la severidad, que puede ser: bloqueante, crítica, mayor, menor o informativa.

Por defecto, Sonar utiliza para java el perfil denominado Sonar way. En la versión 9.9.3 de SonarQube, este perfil se compone de un total de 477 reglas.

Desde versiones anteriores ya se había mejorado la definición de las reglas, revisándose la severidad de las mismas y agrupándolas en categorías. Con esta redefinición se ha buscado el dar un tratamiento diferenciado a las incidencias relacionadas con la fiabilidad del código (bugs) o con la seguridad del mismo (vulnerabilidades), del resto de incidencias que pueden afectar a la mantenibilidad (agrupadas bajo la denominación genérica de code smell). Por otra parte, para determinar la severidad de cada regla, se ha utilizado la siguiente tabla:

	Impacto	Probabilidad
Bloqueante	Alto	Alta
Crítica	Alto	Baja
Mayor	Bajo	Alta
Menor	Bajo	Baja

Para el perfil Sonar way para Java, las reglas se categorizan de la siguiente manera:

- 139 Bugs (12 Bloqueantes, 20 Críticas , 87 Mayores, 20 Menores y 0 Informativas), relacionados con la fiabilidad del código.
- 31 Vulnerabilidades (4 Bloqueantes, 19 Críticas, 6 Mayores, 2 Menores y 0 Informativas), relacionados con la seguridad.
- 270 Code Smells (14 Bloqueantes, 40 Críticas, 112 Mayores, 100 Menores y 4 Informativas), relacionados con la mantenibilidad.
- 37 Hotspot Seguridad (2 Bloqueantes, 14 Críticas, 13 Mayores , 8 Menores y 0 Informativas)

Con el fin de ajustar el perfil por defecto a las necesidades de la Consejería, se ha definido un perfil especifico, denominado **CMAOT**, en el cual se ha adaptado el perfil Sonar way a las especificaciones del Libro Blanco de Desarrollo.

- Se ha modificado la regla “Package names should comply with a naming convention”, de manera que la nomenclatura de los paquetes se adapte a la indicada en el Libro Blanco. Esta regla está clasificada como “menor”:
  - *^es.juntadeandalucia.cma(ot)?+(\.[a-z][a-z0-9]*)*$*

En la versión del perfil de la Consejería, **CMAOT_02.00**, se han definido un total de 32 reglas bloqueantes:

- 88 Bugs, relacionados con la fiabilidad del código.
- 15 Vulnerabilidades, relacionados con la seguridad.
- 139 Code Smells, relacionados con la mantenibilidad.

Aunque en el APÉNDICE I se describen las incidencias catalogadas como bloqueantes, dentro de la propia herramienta SonarQube puede encontrarse una descripción más detallada de cada regla, así como ejemplos del error y posibles alternativas para su subsanación.

3. Cuadro de Mando. Métricas

El cuadro de mando de Sonar refleja en una pantalla gran cantidad de información que nos permite conocer la calidad del código de una aplicación. Esta información es posible personalizarla de acuerdo con las necesidades de cada instalación.

SonarQube proporciona un conjunto de métricas relacionadas con la calidad y mantenibilidad del código, entre las cuales destacan:

- Tamaño de la aplicación
- Grado de documentación del código
- Grado de duplicidad de código
- Complejidad
- Deuda técnica y ratio de deuda técnica
- Número de defectos, vulnerabilidades y code smells
- Seguridad
- Cobertura
- Mantenibilidad

SonarQube permite controlar la evolución de la calidad de las diferentes versiones de una aplicación. Con ello, es posible conocer la calidad inicial de una aplicación y detectar si una aplicación va degradando su diseño inicial o si se van realizando correcciones sobre los parámetros que resultan en valores insatisfactorios de la calidad global del código.

Para ello, es posible comparar los valores de las métricas entre diferentes entregas del software. Esto se realiza desde el panel de una aplicación o desde la opción de comparación de proyectos, que permite tanto comparar métricas de diferentes proyectos como versiones de un mismo proyecto.

El estudio de la evolución de los diferentes parámetros que proporciona SonarQube facilita la adopción de acciones correctoras que eviten que la calidad del código disminuya.

Los parámetros objetivo que requiere la Consejería para dar por correcta una entrega son las documentadas en el Umbral (Quality Gate) **CMAOT QG_2.1.0**.

El tratamiento de la calidad de una aplicación será diferente según se trate de una aplicación nueva o de un mantenimiento. En el primer caso, se requerirá el cumplimiento de valores umbrales de los parámetros, mientras que en el segundo, se buscará la mejora de los parámetros actuales.

En ningún caso, la subida a producción de una aplicación o actualización catalogada como crítica será bloqueada por el incumplimiento de alguno de las ratios anteriores; siempre dependiendo de la decisión del Director de Proyecto y Jefatura

4. Parámetros Objetivo en la Consejería. Umbrales (Quality Gates)

El umbral de Calidad a usar en el análisis de las aplicaciones es CMAOT QG_2.1.0, cuyos valores de referencia son los que se resumen en el siguiente cuadro y **han sido adaptados a la normativa actual de la ADA**:

4.1. Umbral Parcial Sistema Información Legacy

- Tiene como objetivo evaluar la calidad del código fuente en el nuevo código implementado. Será lanzado en cada nueva release de un componente
- *Obligatorio*, *Recomendado*, *Fallo*

Métrica	Valor Objetivo
New blocker issues	***0-5*** ***>5*** No permite nuevos issue (bug, code smells y vulnerability) con severidad bloqueante
Maintainability rating	***A o B*** ***C*** ***D o E*** La calificación otorgada a su proyecto en relación con el valor de su índice de deuda técnica . La cuadrícula de calificación de mantenibilidad predeterminada es: A=0 - 0,05 ; B=0,06 – 0,1 ; C=0,11 - 0,20 ; D=0,21 – 0,5 ; E=0,51 - 1 La escala de calificación de Mantenibilidad se puede expresar alternativamente diciendo que si el costo de remediación pendiente es: * <=5% del tiempo que ya ha invertido en la aplicación, la calificación es A * entre 6 a 10% la calificación es una B * entre 11 a 20% la calificación es una C * entre 21 a 50% la calificación es una D * cualquier valor superior al 50% es una E

Las pautas que deben seguirse, al realizarse una nueva entrega, de un Sistema de Información Legacy:

- El Proveedor debe conocer estos umbrales establecidos
- Análisis previo por el equipo de Calidad, sin que venga acompañado de entrega de Software o sobre la última versión desplegada en el entorno de Pruebas.

- Aplicar Umbral Parcial Legacy, en el que si hay incumplimiento de dichas métricas, el equipo de Calidad lo bloqueará hasta que sea corregido.

- El umbral del Sistema Información Nuevo se emplearán desde la tercera entrega

4.2. Umbral Total Sistema Información Nuevo

- Tiene como objetivo evaluar la calidad del código fuente total y debe llevarse a cabo en las versiones iniciales del componente
- *Obligatorio*, *Recomendado*, *Fallo*

Métrica	Valor Objetivo
Total Blocker issues	***0-5*** ***>5*** Total de issue (bug, code smells y vulnerability) con severidad bloqueante
Total Comments (%)	***>10%*** ***5%-10%*** ***<5%*** La densidad de líneas de comentarios = líneas de comentarios / (líneas de código + líneas de comentarios) * 100 Con tal fórmula: * 50% significa que el número de líneas de código es igual al número de líneas de comentarios * 100% significa que el archivo solo contiene líneas de comentarios
Total Maintainability rating	***A o B*** ***C*** ***D o E*** La calificación otorgada a su proyecto en relación con el valor de su índice de deuda técnica . La cuadrícula de calificación de mantenibilidad predeterminada es: A=0 - 0,05 ; B=0,06 – 0,1 ; C=0,11 - 0,20 ; D=0,21 – 0,5 ; E=0,51 – 1 La escala de calificación de Mantenibilidad se puede expresar alternativamente diciendo que si el costo de remediación pendiente es: * <=5% del tiempo que ya ha invertido en la aplicación, la calificación es A * entre 6 a 10% la calificación es una B * entre 11 a 20% la calificación es una C * entre 21 a 50% la calificación es una D * cualquier valor superior al 50% es una E
Total Reliability Rating	***A o B*** ***C*** ***D o E*** A = 0 errores B = al menos 1 error menor C = al menos 1 error mayor D = al menos 1 error crítico E = al menos 1 error bloqueador

Métrica	Valor Objetivo
Total Security Rating	***A o B*** ***C*** ***D o E*** A = 0 Vulnerabilidades B = al menos 1 Vulnerabilidad Menor C = al menos 1 Vulnerabilidad Mayor D = al menos 1 Vulnerabilidad Crítica E = al menos 1 Vulnerabilidad de Bloqueador
Total % Lineas duplicadas	***<10%*** ***10%-15%*** ***>15%*** Porcentaje de código que está duplicado
Total % de Ejecución de Test Unitarios	***>50%*** ***0-50%*** Porcentaje de test del total del proyecto que se ejecutaron con éxito
Total Cobertura del código	***>80%*** ***70-80%*** ***<70%*** Tiene como objetivo proporcionar una visión sobre cuánto código está cubierto por pruebas unitarias
Total Bugs	***0-5*** ***>5*** Error de codificación que hace el sistema vulnerable a posibles defectos y que deben ser corregidos inmediatamente
Total Vulnerabilidades	***0-5*** ***>5*** Número de vulnerabilidades detectadas

Las pautas que deben seguirse al realizarse una entrega, de un Sistema de Información Nuevo, son:

- En la primera versión o primera ocurrencia, si se producen incumplimientos del Umbral Total, el equipo de Calidad informará al Proveedor y Director de Proyecto del resultado del análisis y se solicitará que en la siguiente versión del componente, los umbrales sean acordes a estos establecidos
    - Se generará para ello, una Mantis Menor.

- En la siguiente versión del componente o segunda ocurrencia, debe existir una Mantis menor y que se haya producido un nuevo incumplimiento del Umbral Total; el equipo de Calidad informará al Proveedor y Director de Proyecto del resultado y se solicitará corrección.
    - Se generará para ello, una Mantis bloqueante.

5. Procedimiento de Revisión y Control de Calidad del Código

5.1. Actuaciones previas por el Proveedor

Con el fin de que el código Java que compone la entrega cumpla con los requisitos de calidad establecidos por la Consejería y de esta manera minimizar el número de rechazos, estará disponible en el Portal del Conocimiento el conjunto de reglas de calidad aplicables al código Java. De esta manera, el proveedor podrá tener conocimiento previo del nivel de calidad de su código y tomar las medidas necesarias para evitar rechazos.

La Consejería facilitará a los proveedores el conjunto de reglas y umbrales de calidad esperados, de manera que puedan realizar las comprobaciones necesarias antes de proceder a la entrega del desarrollo.

5.2. Respuesta Inmediata

Una vez que el proveedor haya realizado la entrega del software, la Oficina de Calidad, dentro de las comprobaciones que forman parte de la Respuesta Inmediata, realizará una revisión del código haciendo uso de SonarQube.

El resultado de la revisión, si se detectaran defectos, se registrará en el gestor de incidencias. Para ello, se registrará una incidencia del tipo “Verificación de Entrega” (“Generación Sonar”). Esta incidencia no será en ningún caso bloqueante si la entrega fuese crítica. Para el resto se seguirá el criterio expresado en el capítulo anterior.

El proveedor podrá contestar la incidencia en Mantis abierta añadiendo sus comentarios y, en el caso de que en su opinión se tratase de un falso positivo, incluirá sus alegaciones. La Oficina de Calidad analizará la respuesta y, si estuviese de acuerdo con las justificaciones aportadas, podrá marcar en SonarQube la incidencia como falso positivo y cerrar la incidencia si procede.

6. APÉNDICE 1. Descripción de las reglas javas bloqueantes

Como se indicó en el apartado 2, en el perfil CMAOT se incluyen un total de 32 incidencias bloqueantes, divididas en:

- Vulnerabilidad: 4
- Bugs: 12
- Code Smell: 14
- Hotspots de Seguridad: 2

Las clases con "@Controller" que usan "@SessionAttributes" deben llamar a "setComplete" en sus objetos "SessionStatus"	
Problema	Un @Controller de Spring que usa @SessionAttributes está diseñado para manejar un formulario con estado / de múltiples envíos. Tales @Controllers utilizan los @SessionAttributes especificados para almacenar datos en el servidor entre solicitudes. Esos datos deben limpiarse cuando la sesión termine, pero a menos que se llame a setComplete() en el objeto SessionStatus desde un método @RequestMapping, ni Spring ni la JVM sabrán que es el momento de hacerlo. Ten en cuenta que el objeto SessionStatus debe ser pasado como parámetro a ese método.

Las clases con "@Controller" que usan "@SessionAttributes" deben llamar a "setComplete" en sus objetos "SessionStatus"	

Corrección	Erróneo:
	@Controller
	@SessionAttributes("hello") // Noncompliant; this doesn't get cleaned up
	public class HelloWorld {
	
	  @RequestMapping("/greet", method = GET)
	  public String greet(String greetee) {
	
	  return "Hello " + greetee;
	  }
	}
	
	Correcto:
	@Controller
	@SessionAttributes("hello")
	public class HelloWorld {
	
	  @RequestMapping("/greet", method = GET)
	  public String greet(String greetee) {
	
	  return "Hello " + greetee;
	  }
	
	  @RequestMapping("/goodbye", method = POST)
	  public String goodbye(SessionStatus status) {
	  //...
	  status.setComplete();
	  }
	
	}

Las clases con "@Controller" que usan "@SessionAttributes" deben llamar a "setComplete" en sus objetos "SessionStatus"	
Excepciones admitidas	Ninguna
Clasificación	Bug

No se deben usar "@SpringBootApplication" y "@ComponentScan" en el paquete predeterminado	
Problema	Se usa @ComponentScan para determinar qué Beans de Spring están disponibles en el contexto de la aplicación. Los paquetes a escanear pueden configurarse mediante los parámetros basePackageClasses o basePackages (o su valor alias). Si ninguno de los parámetros está configurado, @ComponentScan solo considerará el paquete de la clase anotada con ella. Cuando @ComponentScan se usa en una clase que pertenece al paquete predeterminado, se escaneará todo el classpath. Esto ralentizará el inicio de la aplicación y es probable que la aplicación no se inicie debido a una BeanDefinitionStoreException, ya que acabarías escaneando el propio paquete del Framework de Spring. Esta regla plantea un problema cuando: • @ComponentScan, @SpringBootApplication y @ServletComponentScan se usan en una clase que pertenece al paquete predeterminado. • @ComponentScan se configura explícitamente con el paquete por defecto

No se deben usar "@SpringBootApplication" y "@ComponentScan" en el paquete predeterminado	

Corrección	Erróneo:
	import org.springframework.boot.SpringApplication;
	
	@SpringBootApplication // Noncompliant; RootBootApp is declared in the default package
	public class RootBootApp {
	...
	}
	
	@ComponentScan("")
	public class Application {
	...
	}
	
	Correcto:
	package hello;
	
	import org.springframework.boot.SpringApplication;
	
	@SpringBootApplication // Compliant; RootBootApp belongs to the "hello" package
	public class RootBootApp {
	...
	}

Excepciones admitidas	Ninguna
Clasificación	Bug

El método “clone” no debe ser sobrescrito	
Problema	Muchos consideran que clone y Cloneable están rotos en Java, en gran parte porque las reglas para sobrescribir clone son complicadas y difíciles de hacer correctamente, según Joshua Bloch: “El método clone de Object es muy complicado. Se basa en copias de campos, y es 'extra-lingüístico'. Crea un objeto sin llamar a un constructor. No hay garantías de que preserve las invariantes establecidas por los constructores. A lo largo de los años ha habido muchos errores, tanto dentro como fuera de Sun, derivados del hecho de que si simplemente llamas a super.clone repetidamente en la cadena hasta que hayas clonado un objeto, obtienes una copia superficial del objeto. Generalmente, el clon comparte el estado con el objeto que se está clonando. Si ese estado es mutable, no tienes dos objetos independientes. Si modificas uno, el otro también cambia. Y de repente, obtienes un comportamiento aleatorio.” En su lugar, se debe utilizar un constructor de copia o una fábrica de copias. Esta regla plantea un problema cuando se sobrescribe clone, independientemente de si Cloneable está implementado o no.

Corrección	Erróneo:
	public class MyClass {
	// ...
	
	public Object clone() { // Noncompliant
	//...
	}
	}
	
	Correcto:
	public class MyClass {
	// ...
	
	MyClass (MyClass source) {
	//...
	}
	}

Excepciones admitidas	Ninguna
Clasificación	Code Smell

Los métodos "PreparedStatement" y "ResultSet" deben ser llamados con índices válidos	
Problema	Los parámetros en un PreparedStatement están indexados comenzando desde 1, no desde 0, por lo que usar cualquier método 'set' de un PreparedStatement con un número menor que 1 es un error, al igual que usar un índice mayor que el número de parámetros. El mismo estilo de indexación también se aplica a ResultSet.

Corrección	Erróneo:
	PreparedStatement ps = con.prepareStatement("SELECT fname, lname FROM employees where hireDate > ? and salary < ?");
	ps.setDate(0, date); // Noncompliant
	ps.setDouble(3, salary); // Noncompliant
	
	ResultSet rs = ps.executeQuery();
	while (rs.next()) {
	String fname = rs.getString(0); // Noncompliant
	// ...
	}
	
	Correcto:
	PreparedStatement ps = con.prepareStatement("SELECT fname, lname FROM employees where hireDate > ? and salary < ?");
	ps.setDate(1, date);
	ps.setDouble(2, salary);
	
	ResultSet rs = ps.executeQuery();
	while (rs.next()) {
	String fname = rs.getString(1);
	// ...
	}

Excepciones admitidas	Ninguna
Clasificación	Bug

Las sentencias “switch” no deben contener etiquetas que no sean de tipo “case”	
Problema	Incluso si es legal, mezclar etiquetas 'case' y etiquetas no 'case' en el cuerpo de una sentencia 'switch' es muy confuso y puede incluso ser el resultado de un error tipográfico.

Las sentencias “switch” no deben contener etiquetas que no sean de tipo “case”	

Corrección	Erróneo:
	switch (day) {
	case MONDAY:
	case TUESDAY:
	WEDNESDAY: // Noncompliant; syntactically correct, but behavior is not what's expected
	doSomething();
	break;
	...
	}
	
	switch (day) {
	case MONDAY:
	break;
	case TUESDAY:
	foo:for(int i = 0 ; i < X ; i++) { // Noncompliant; the code is correct and behaves as expected but is barely readable
	/* ... */
	break foo; // this break statement doesn't relate to the nesting case TUESDAY
	/* ... */
	}
	break;
	/* ... */
	}
	
	Correcto:
	switch (day) {
	case MONDAY:
	case TUESDAY:
	case WEDNESDAY:
	doSomething();
	break;

Las sentencias “switch” no deben contener etiquetas que no sean de tipo “case”	

	...
	}
	
	switch (day) {
	case MONDAY:
	break;
	case TUESDAY:
	compute(args); // put the content of the labelled "for" statement in a dedicated method
	break;
	
	/* ... */
	}

Excepciones admitidas	Ninguna
Clasificación	Code Smell

No debe utilizarse “ThreadGroup”	
Problema	Hay pocas razones válidas para utilizar los métodos de la clase ThreadGroup. Algunos están obsoletos (allowThreadSuspension(), resume(), stop() y suspend()), otros son obsoletos, algunos no son seguros para hilos y otros son inseguros (activeCount(), enumerate()).
Corrección	Cualquier uso de ThreadGroup es sospechoso y debe ser evitado.
Excepciones admitidas	Ninguna
Clasificación	Code Smell

No se debe llamar a “wait” cuando se mantienen múltiples bloqueos	
Problema	Cuando se mantienen dos bloqueos simultáneamente, una llamada a 'wait' solo libera uno de ellos. El otro se mantendrá hasta que otro hilo solicite un bloqueo sobre el objeto esperado. Si ningún otro código no relacionado intenta bloquear ese objeto, todos los demás hilos quedarán bloqueados, lo que resultará en un bloqueo mutuo.
Corrección	Cualquier uso de ThreadGroup es sospechoso y debe ser evitado.
Excepciones admitidas	Ninguna
Clasificación	Code Smell

No se debe llamar a “wait” cuando se mantienen múltiples bloqueos	
Problema	Cuando se mantienen dos bloqueos simultáneamente, una llamada a 'wait' solo libera uno de ellos. El otro se mantendrá hasta que otro hilo solicite un bloqueo sobre el objeto esperado. Si ningún otro código no relacionado intenta bloquear ese objeto, todos los demás hilos quedarán bloqueados, lo que resultará en un bloqueo mutuo.

Corrección	Erróneo:
	synchronized (this.mon1) { // threadB can't enter this block to request this.mon2 lock & release threadA
	synchronized (this.mon2) {
	this.mon2.wait(); // Noncompliant; threadA is stuck here holding lock on this.mon1
	}
	}

Excepciones admitidas	Ninguna
Clasificación	Bug

“wait(...)” debe usarse en lugar de “Thread.sleep(...)” cuando se mantiene un bloqueo	
Problema	Si se llama a Thread.sleep(...) cuando el hilo actual tiene un bloqueo, podría generar problemas de rendimiento y escalabilidad, o incluso peores consecuencias como bloqueos mutuos, ya que la ejecución del hilo que mantiene el bloqueo queda congelada.

“wait(...)” debe usarse en lugar de “Thread.sleep(...)” cuando se mantiene un bloqueo	
Corrección	Llamar a wait(...) en el objeto monitor para liberar temporalmente el bloqueo y permitir que otros hilos se ejecuten.
Excepciones admitidas	Ninguna
Clasificación	Bug

Se debe usar una contraseña segura al conectarse a una base de datos	
Problema	Cuando se depende del modo de autenticación por contraseña para la conexión a la base de datos, se debe elegir una contraseña segura. Esta regla plantea un problema cuando se utiliza una contraseña vacía.
Corrección	Erróneo: Connection conn = DriverManager.getConnection("jdbc:derby:memory:myDB;create=true", "login", ""); Correcto: String password = System.getProperty("database.password"); Connection conn = DriverManager.getConnection("jdbc:derby:memory:myDB;create=true", "login", password);
Excepciones admitidas	Ninguna
Clasificación	Vulnerabilidad

Las aserciones deben ser completas	
Problema	Es muy fácil escribir aserciones incompletas cuando se utilizan algunos marcos de prueba. Esta regla fuerza aserciones completas en los siguientes casos: • Fest: assertThat no es seguido por una invocación de aserción. • AssertJ: assertThat no es seguido por una invocación de aserción. • Mockito: verify no va seguido de una invocación a un método. • Truth: assertXXX no va seguida de una invocación a una aserción. En tales casos, lo que se supone que es una prueba no verifica realmente nada.

Las aserciones deben ser completas	

Corrección	Erróneo:
	// Fest
	boolean result = performAction();
	// let's now check that result value is true
	assertThat(result); // Noncompliant; nothing is actually checked, the test passes whether "result" is true or false
	
	// Mockito
	List mockedList = Mockito.mock(List.class);
	mockedList.add("one");
	mockedList.clear();
	// let's check that "add" and "clear" methods are actually called
	Mockito.verify(mockedList); // Noncompliant; nothing is checked here, oups no call is chained to verify()
	
	Correcto:
	// Fest
	boolean result = performAction();
	// let's now check that result value is true
	assertThat(result).isTrue();
	
	// Mockito
	List mockedList = Mockito.mock(List.class);
	mockedList.add("one");
	mockedList.clear();
	// let's check that "add" and "clear" methods are actually called
	Mockito.verify(mockedList).add("one");
	Mockito.verify(mockedList).clear();

Las aserciones deben ser completas	

Excepciones admitidas	Las asignaciones de variables y las sentencias de retorno se omiten para permitir métodos auxiliares.
	private BooleanAssert check(String filename, String key) {
	String fileContent = readFileContent(filename);
	performReplacements(fileContent);
	return assertThat(fileContent.contains(key)); // No issue is raised here
	}
	
	@Test
	public void test() {
	check("foo.txt", "key1").isTrue();
	check("bar.txt", "key2").isTrue();
	}

Clasificación	Code smell

Los campos de la clase hija no deben ocultar los campos de la clase padre	
Problema	Sobrescribir el método Object.finalize() para disponer de los recursos del sistema es algo que debe hacerse con cuidado. Se recomienda llamar a super.finalize() al final de la implementación del método por si las implementaciones del padre también deben liberar recursos del sistema.

Los campos de la clase hija no deben ocultar los campos de la clase padre	

Corrección	Erróneo:
	public class Fruit {
	  protected Season ripe;
	  protected Color flesh;
	
	  // ...
	}
	
	public class Raspberry extends Fruit {
	  private boolean ripe; // Noncompliant
	  private static Color FLESH; // Noncompliant
	}
	
	Correcto:
	public class Fruit {
	  protected Season ripe;
	  protected Color flesh;
	
	  // ...
	}
	
	public class Raspberry extends Fruit {
	  private boolean ripened;
	  private static Color FLESH_COLOR;
	
	}

Los campos de la clase hija no deben ocultar los campos de la clase padre	

Excepciones admitidas	Esta regla ignora los campos con el mismo nombre que son estáticos tanto en la clase padre como en la clase hija. Esta regla ignora los campos privados de la clase padre, pero en todos los demás casos, el campo de la clase hija debe ser renombrado.
	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
	// ...
	}

Clasificación	Code Smell

Las credenciales no deben estar codificadas de forma fija	
Problema	Se ha encontrado una clave codificada de forma fija en tu código. Debes listar rápidamente dónde se utiliza esta clave, revocarla y luego cambiarla en cada sistema que lo utilice. Las contraseñas, claves y cualquier tipo de credenciales solo deben utilizarse para autenticar una sola entidad (una persona o un sistema). Si permites que terceros se autentiquen como otro sistema o persona, pueden suplantar identidades legítimas y socavar la confianza dentro de la organización. No importa si la suplantación es maliciosa: en cualquiera de los casos, es una clara violación de confianza en el sistema, ya que los sistemas involucrados asumen falsamente que la entidad autenticada es quien dice ser. Las consecuencias pueden ser catastróficas. Mantener credenciales en texto plano en una base de código es equivalente a compartir esa contraseña con cualquiera que tenga acceso al código fuente y a los servidores en tiempo de ejecución. Por lo tanto, es una violación de confianza, ya que estas personas tienen la capacidad de suplantar a otros. Los servicios de gestión de claves son las herramientas más eficientes para almacenar credenciales y proteger las identidades asociadas con ellas. Los proveedores de la nube y los servicios locales pueden utilizarse para este propósito. Si no es posible almacenar credenciales en un servicio de gestión de datos secretos, sigue estas pautas: • No almacenes credenciales en un archivo al que pueda acceder un número excesivo de personas. ◦ Por ejemplo, no en el código, no en una hoja de cálculo, no en una nota adhesiva y no en una unidad compartida. • Utiliza el sistema operativo de producción para proteger el control de acceso a las contraseñas. ◦ Por ejemplo, en un archivo cuyos permisos estén restringidos y protegidos con chmod y chown.

Corrección	Erróneo:
	
	import org.h2.security.SHA256;
	
	String inputString = "s3cr37";
	byte[] key = inputString.getBytes();
	
	SHA256.getHMAC(key, message); // Noncompliant

Las credenciales no deben estar codificadas de forma fija	

	Correcto:
	AWS Secrets Manager:
	import software.amazon.awssdk.services.secretsmanager.SecretsManagerClient;
	import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
	import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;
	import org.h2.security.SHA256;
	
	public static void doSomething(SecretsManagerClient secretsClient, String secretName) {
	GetSecretValueRequest valueRequest = GetSecretValueRequest.builder()
	.secretId(secretName)
	.build();
	
	GetSecretValueResponse valueResponse = secretsClient.getSecretValue(valueRequest);
	String secret = valueResponse.secretString();
	
	byte[] key = secret.getBytes();
	SHA256.getHMAC(key, message);
	}
	
	Azure Key Vault Secret:
	import com.azure.identity.DefaultAzureCredentialBuilder;
	import com.azure.security.keyvault.secrets.SecretClient;
	import com.azure.security.keyvault.secrets.SecretClientBuilder;
	import com.azure.security.keyvault.secrets.models.KeyVaultSecret;
	import org.h2.security.SHA256;
	
	public static void doSomething(SecretClient secretClient, String secretName) {
	KeyVaultSecret retrievedSecret = secretClient.getSecret(secretName);
	String secret = retrievedSecret.getValue();

Las credenciales no deben estar codificadas de forma fija	
Excepciones admitidas	Ninguna
Clasificación	Vulnerabilidad

No se debe usar el bloqueo de doble comprobación	
Problema	El bloqueo de doble comprobación es la práctica de verificar el estado de un objeto de inicialización perezosa tanto antes como después de entrar en un bloque sincronizado para determinar si se debe o no inicializar el objeto. No funciona de manera confiable e independiente de la plataforma sin una sincronización adicional para las instancias mutables de cualquier cosa que no sea float o int. Usar bloqueo de doble comprobación para la inicialización perezosa de cualquier otro tipo de objeto primitivo o mutable corre el riesgo de que un segundo hilo use un miembro no inicializado o parcialmente inicializado mientras el primer hilo aún lo está creando, lo que podría hacer que el programa se bloquee. Existen varias formas de solucionar esto. La más simple es no usar el bloqueo de doble comprobación en absoluto, y sincronizar todo el método en su lugar. Con versiones anteriores de la JVM, se solía desaconsejar la sincronización del método completo por razones de rendimiento. Sin embargo, el rendimiento de la sincronización ha mejorado mucho en las versiones más recientes de la JVM, por lo que ahora es una solución preferida. Si prefieres evitar usar sincronización por completo, puedes usar una clase estática interna para mantener la referencia. Las clases estáticas internas se cargan de manera perezosa.

No se debe usar el bloqueo de doble comprobación	

Corrección	Erróneo:
	@NotThreadSafe
	public class DoubleCheckedLocking {
	  private static Resource resource;
	
	  public static Resource getInstance() {
	  if (resource == null) {
	  synchronized (DoubleCheckedLocking.class) {
	  if (resource == null)
	  resource = new Resource();
	  }
	  }
	  return resource;
	  }
	
	  static class Resource {
	  }
	}
	
	Correcto:
	@ThreadSafe
	public class SafeLazyInitialization {
	  private static Resource resource;
	
	  public static synchronized Resource getInstance() {
	  if (resource == null)
	  resource = new Resource();
	  return resource;
	  }
	  static class Resource {
	  }

No se debe usar el bloqueo de doble comprobación	

```
}
Con un holder estático interno:
@ThreadSafe
public class ResourceFactory {
   private static class ResourceHolder {
      public static Resource resource = new Resource(); // This will be lazily initialised
   }

   public static Resource getResource() {
      return ResourceFactory.ResourceHolder.resource;
   }

   static class Resource {
   }
}

Usando 'volatile':
class ResourceFactory {
  private volatile Resource resource;

  public Resource getResource() {
    Resource localResource = resource;
    if (localResource == null) {
      synchronized (this) {
        localResource = resource;
        if (localResource == null) {
          resource = localResource = new Resource();
        }
      }
    }
    return localResource;
  }
```

No se debe usar el bloqueo de doble comprobación	
Excepciones admitidas	Ninguna
Clasificación	Bug

No se deben usar archivos abiertos en modo de adición (append) con ObjectOutputStream	
Problema	ObjectOutputStreams se usan con la serialización, y lo primero que escribe un ObjectOutputStream es el encabezado del flujo de serialización. Este encabezado debe aparecer una sola vez por archivo, al principio. Si pasas un archivo abierto en modo de adición (append) al constructor de un ObjectOutputStream, el encabezado del flujo de serialización se añadirá al final del archivo antes de que tu objeto también se agregue. Cuando intentes leer tu objeto(s) de vuelta desde el archivo, solo el primero será leído correctamente, y luego se lanzará una excepción StreamCorruptedException.

Corrección	Erróneo:
	FileOutputStream fos = new FileOutputStream (fileName , true); // fos opened in append mode
	ObjectOutputStream out = new ObjectOutputStream(fos); // Noncompliant
	
	Correcto:
	FileOutputStream fos = new FileOutputStream (fileName);
	ObjectOutputStream out = new ObjectOutputStream(fos);

Excepciones admitidas	Ninguna
Clasificación	Bug

<table>
<tr><th colspan="2">No se deben usar palabras clave reservadas como nombres</th></tr>
<tr><td>Problema</td><td>A lo largo de la evolución de Java se han añadido palabras clave. Mientras que el código que usa esas palabras como identificadores puede ser compilable en versiones anteriores de Java, no lo será en versiones modernas.

Las siguientes palabras clave están marcadas como identificadores no válidos.

<table><tr><th>Palabra clave</th><th>Añadido</th></tr><tr><td>_</td><td>9</td></tr><tr><td>enum</td><td>5.0</td></tr></table>
assert y strictfp son otro ejemplo de identificadores válidos que se convirtieron en palabras clave en versiones posteriores, pero que no son compatibles con esta regla.</td></tr>
<tr><td>Corrección</td><td>Erróneo:

public void doSomething() {
int enum = 42; // Noncompliant
String _ = ""; // Noncompliant
}

Correcto:

public void doSomething() {
int magic = 42;
}</td></tr>
<tr><td>Excepciones admitidas</td><td>Ninguna</td></tr>
<tr><td>Clasificación</td><td>Code smell</td></tr>
</table>

Las contraseñas codificadas de forma fija son sensibles a la seguridad	
Problema	Debido a que es fácil extraer cadenas del código fuente o binario de una aplicación, las contraseñas no deben estar codificadas de forma fija. Esto es particularmente cierto para aplicaciones que son distribuidas o de código abierto. En el pasado, esto ha llevado a las siguientes vulnerabilidades: • CVE-2019-13466. • CVE-2018-15389. Las contraseñas deben almacenarse fuera del código, en un archivo de configuración, una base de datos o un servicio de gestión de contraseñas. Esta regla marca las instancias de contraseñas codificadas de forma fija utilizadas en conexiones a bases de datos y LDAP. Busca contraseñas codificadas de forma fija en cadenas de conexión y nombres de variables que coincidan con cualquiera de los patrones de la lista proporcionada.
Corrección	• Almacena las credenciales en un archivo de configuración que no se suba al repositorio de código. • Almacena las credenciales en una base de datos. • Utiliza el servicio de tu proveedor de nube para gestionar secretos. • Si una contraseña ha sido revelada a través del código fuente: cámbiala
Excepciones admitidas	Ninguna
Clasificación	Hotspots de Seguridad

Las claves codificadas de manera fija son sensibles a la seguridad	
Problema	Debido a que es fácil extraer cadenas de texto del código fuente o binario de una aplicación, las claves no deben estar codificadas de manera fija. Esto es especialmente cierto para aplicaciones que son distribuidas o de código abierto. En el pasado, esto ha llevado a las siguientes vulnerabilidades: • CVE-2022-25510. • CVE-2021-42635. Las claves deben almacenarse fuera del código fuente, en un archivo de configuración o en un servicio de gestión de claves. Esta regla detecta variables/campos cuyo nombre coincide con una lista de palabras (secret, token, credential, auth, api[_.-]?key) y que se les asigna un valor pseudorandom codificado de manera fija. La pseudorandomicidad del valor codificado de manera fija se basa en su entropía y la probabilidad de que sea legible por humanos. La sensibilidad de la aleatoriedad puede ajustarse si es necesario. Valores más bajos detectarán valores menos aleatorios, lo que puede generar más falsos positivos.
Corrección	Almacena la clave en un archivo de configuración que no se suba al repositorio de código. Utiliza el servicio de tu proveedor de la nube para gestionar claves. Si una clave ha sido revelada a través del código fuente: revócala y crea una nueva.
Excepciones admitidas	Ninguna
Clasificación	Hotspots de Seguridad

Los casos de prueba de JUnit deben llamar a los métodos de la clase padre	
Problema	Sobrescribir un método de la clase padre impide que ese método sea llamado, a menos que se haga una llamada explícita a super en el método que lo sobrescribe. En algunos casos, no llamar al método super es aceptable, pero no con setUp y tearDown en un TestCase de JUnit 3.

Los casos de prueba de JUnit deben llamar a los métodos de la clase padre	

Corrección	Erróneo:
	public class MyClassTest extends MyAbstractTestCase {
	
	private MyClass myClass;
	@Override
	protected void setUp() throws Exception { // Noncompliant
	myClass = new MyClass();
	}
	
	Correcto:
	public class MyClassTest extends MyAbstractTestCase {
	
	private MyClass myClass;
	@Override
	protected void setUp() throws Exception {
	super.setUp();
	myClass = new MyClass();
	}

Excepciones admitidas	Ninguna
Clasificación	Code Smell

Los bucles no deben ser infinitos	
Problema	Un bucle infinito es aquel que nunca termina mientras el programa está en ejecución, es decir, tienes que terminar el programa para salir del bucle. Ya sea al cumplir la condición de fin del bucle o mediante un break, todo bucle debe tener una condición de fin.

Los bucles no deben ser infinitos	

Corrección	Erróneo:
	for (;;) { // Noncompliant; end condition omitted
	// ...
	}
	
	int j;
	while (true) { // Noncompliant; end condition omitted
	j++;
	}
	
	int k;
	boolean b = true;
	while (b) { // Noncompliant; b never written to in loop
	k++;
	}
	
	Correcto:
	int j;
	while (true) { // reachable end condition added
	j++;
	if (j == Integer.MIN_VALUE) { // true at Integer.MAX_VALUE +1
	break;
	}
	}
	
	int k;
	boolean b = true;
	while (b) {
	k++;
	b = k < Integer.MAX_VALUE;
	}

Los bucles no deben ser infinitos	
Excepciones admitidas	Ninguna
Clasificación	Bug

No se deben llamar los métodos 'wait(...)', 'notify()' y 'notifyAll()' en instancias de Thread	
Problema	Los métodos wait(...), notify() y notifyAll() están disponibles en una instancia de Thread, pero solo porque todas las clases en Java extienden de Object y, por lo tanto, heredan automáticamente esos métodos. Pero hay dos muy buenas razones para no llamarlos en un Thread: Internamente, la JVM depende de estos métodos para cambiar el estado del Thread (BLOQUEADO, ESPERANDO, …), por lo que llamarlos corromperá el comportamiento de la JVM. No está claro (quizás ni siquiera para el programador original) qué se espera realmente. Por ejemplo, ¿se está esperando que la ejecución del Thread se suspenda, o es la adquisición del monitor del objeto lo que se está esperando?
Corrección	Erróneo: Thread myThread = new Thread(new RunnableJob()); ... myThread.wait(2000);
Excepciones admitidas	Ninguna
Clasificación	Bug

Los nombres de los métodos y los campos no deben ser iguales ni diferir solo por la capitalización	
Problema	Al observar el conjunto de métodos en una clase, incluidos los métodos de la clase base, y encontrar dos métodos o campos que solo difieren en la capitalización, esto resulta confuso para los usuarios de la clase. De manera similar, es confuso tener un método y un campo que solo difieren en la capitalización o un método y un campo con el mismo nombre y visibilidad. En el caso de los métodos, podría haber sido un error del desarrollador original, quien tenía la intención de sobrescribir un método de la clase base, pero en su lugar agregó un nuevo método con un nombre casi idéntico. De lo contrario, esta situación simplemente indica una mala elección de nombres. Los nombres de los métodos deben estar orientados a la acción, por lo que deben contener un verbo, lo cual es poco probable en el caso de que tanto un método como un miembro tengan el mismo nombre (con o sin diferencias de capitalización). Sin embargo, cambiar el nombre de un método público podría ser disruptivo para quienes lo llaman. Por lo tanto, la acción recomendada es cambiar el nombre del miembro.

Los nombres de los métodos y los campos no deben ser iguales ni diferir solo por la capitalización	

Corrección	Erróneo:
	public class Car{
	
	public DriveTrain drive;
	
	public void tearDown(){...}
	
	public void drive() {...} // Noncompliant; duplicates field name
	}
	
	public class MyCar extends Car{
	public void teardown(){...} // Noncompliant; not an override. It it really what's intended?
	
	public void drivefast(){...}
	
	public void driveFast(){...} //Huh?
	}
	
	Correcto:
	public class Car{
	
	private DriveTrain drive;
	
	public void tearDown(){...}
	
	public void drive() {...} // field visibility reduced
	}
	
	public class MyCar extends Car{
	@Override
	public void tearDown(){...}

Los nombres de los métodos y los campos no deben ser iguales ni diferir solo por la capitalización	
Excepciones admitidas	Ninguna
Clasificación	Code Smell

Los valores de retorno de los métodos no deben ser invariantes	
Problema	Cuando un método está diseñado para devolver un valor invariante, puede ser un mal diseño, pero no debería afectar negativamente el resultado de tu programa. Sin embargo, cuando esto ocurre en todos los caminos a través de la lógica, seguramente es un error. Esta regla plantea un problema cuando un método contiene varias sentencias de retorno que todas devuelven el mismo valor.

Corrección	Erróneo:
	int foo(int a) {
	int b = 12;
	if (a == 1) {
	return b;
	}
	return b; // Noncompliant
	}

Excepciones admitidas	Ninguna
Clasificación	Code smell

<table>
<tr><th colspan="2">Los métodos no deben llamar a métodos de la misma clase con valores incompatibles de "@Transactional"</th></tr>
<tr><td>Problema</td><td>Cuando se usan proxies de Spring, llamar a un método dentro de la misma clase (por ejemplo, this.aMethod()) con un requisito @Transactional incompatible resultará en excepciones en tiempo de ejecución, porque Spring solo 've' al llamador y no toma medidas para invocar correctamente al receptor.

Por lo tanto, ciertas llamadas nunca deben realizarse dentro de la misma clase:

<table>
<tr><th>Desde</th><th>Hacia</th></tr>
<tr><td>non-@Transactional</td><td>MANDATORY, NESTED, REQUIRED, REQUIRES_NEW</td></tr>
<tr><td>MANDATORY</td><td>NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW</td></tr>
<tr><td>NESTED</td><td>NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW</td></tr>
<tr><td>NEVER</td><td>MANDATORY, NESTED, REQUIRED, REQUIRES_NEW</td></tr>
<tr><td>NOT_SUPPORTED</td><td>MANDATORY, NESTED, REQUIRED, REQUIRES_NEW</td></tr>
<tr><td>REQUIRED or @Transactional</td><td>NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW</td></tr>
<tr><td>REQUIRES_NEW</td><td>NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW</td></tr>
<tr><td>SUPPORTS</td><td>MANDATORY, NESTED, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW</td></tr>
</table>
</td></tr>
</table>

Los métodos no deben llamar a métodos de la misma clase con valores incompatibles de "@Transactional"	

Corrección	Erróneo:
	@Override
	public void doTheThing() {
	// ...
	actuallyDoTheThing(); // Noncompliant
	}
	
	@Override
	@Transactional
	public void actuallyDoTheThing() {
	// ...
	}

Excepciones admitidas	Ninguna
Clasificación	Bug

Las cadenas de formato al estilo printf no deben generar comportamientos inesperados en tiempo de ejecución	
Problema	Debido a que las cadenas de formato al estilo printf se interpretan en tiempo de ejecución, en lugar de ser validadas por el compilador de Java, pueden contener errores que conduzcan a un comportamiento inesperado o a errores en tiempo de ejecución. Esta regla valida estáticamente el buen comportamiento de los formatos al estilo printf al llamar a los métodos format(...) de las clases java.util.Formatter, java.lang.String, java.io.PrintStream, MessageFormat y java.io.PrintWriter, así como los métodos printf(...) de las clases java.io.PrintStream o java.io.PrintWriter.

Las cadenas de formato al estilo printf no deben generar comportamientos inesperados en tiempo de ejecución	

Corrección	Erróneo:
	String.format("The value of my integer is %d", "Hello World"); // Noncompliant; an 'int' is expected rather than a String
	String.format("Duke's Birthday year is %tX", c); //Noncompliant; X is not a supported time conversion character
	String.format("Display %0$d and then %d", 1); //Noncompliant; arguments are numbered starting from 1
	String.format("Not enough arguments %d and %d", 1); //Noncompliant; the second argument is missing
	String.format("%< is equals to %d", 2); //Noncompliant; the argument index '<' refers to the previous format specifier but there isn't one
	
	MessageFormat.format("Result {1}.", value); // Noncompliant; Not enough arguments. (first element is {0})
	MessageFormat.format("Result {{0}.", value); // Noncompliant; Unbalanced number of curly brace (single curly braces should be escaped)
	MessageFormat.format("Result ' {0}", value); // Noncompliant; Unbalanced number of quotes (single quote must be escaped)
	
	java.util.logging.Logger logger;
	logger.log(java.util.logging.Level.SEVERE, "Result {1}!", 14); // Noncompliant - Not enough arguments.
	
	org.slf4j.Logger slf4jLog;
	org.slf4j.Marker marker;
	
	slf4jLog.debug(marker, "message {}"); // Noncompliant - Not enough arguments.
	
	org.apache.logging.log4j.Logger log4jLog;
	log4jLog.debug("message {}"); // Noncompliant - Not enough arguments.
	
	Correcto:
	String.format("The value of my integer is %d", 3);

Las cadenas de formato al estilo printf no deben generar comportamientos inesperados en tiempo de ejecución	

	String.format("Duke's Birthday year is %tY", c);
	String.format("Display %1$d and then %d", 1);
	String.format("Not enough arguments %d and %d", 1, 2);
	String.format("%d is equals to %<", 2);
	
	MessageFormat.format("Result {0}.", value);
	MessageFormat.format("Result {0} & {1}.", value, value);
	MessageFormat.format("Result {0}.", myObject);
	
	java.util.logging.Logger logger;
	logger.log(java.util.logging.Level.SEVERE, "Result {1},{2}!", 14, 2);
	
	org.slf4j.Logger slf4jLog;
	org.slf4j.Marker marker;
	
	slf4jLog.debug(marker, "message {}", 1);
	
	org.apache.logging.log4j.Logger log4jLog;
	log4jLog.debug("message {}", 1);

Excepciones admitidas	Ninguna
Clasificación	Bug

Los recursos deben ser cerrados	
Problema	Las conexiones, flujos, archivos y otras clases que implementan la interfaz Closeable o su superinterfaz, AutoCloseable, deben ser cerradas después de su uso. Además, esa llamada a close debe hacerse en un bloque finally, de lo contrario, una excepción podría impedir que se realice la llamada. Preferentemente, cuando una clase implementa AutoCloseable, el recurso debe ser creado utilizando el patrón 'try-with-resources' y se cerrará automáticamente. No cerrar correctamente los recursos dará lugar a una fuga de recursos, lo que podría hacer que la aplicación y luego quizás el servidor en el que se ejecuta queden completamente afectados.

Corrección	Erróneo:
	
	private void readTheFile() throws IOException {
	Path path = Paths.get(this.fileName);
	BufferedReader reader = Files.newBufferedReader(path, this.charset);
	// ...
	reader.close(); // Noncompliant
	// ...
	Files.lines("input.txt").forEach(System.out::println); // Noncompliant: The stream needs to be closed
	}
	
	private void doSomething() {
	OutputStream stream = null;
	try {
	for (String property : propertyList) {
	stream = new FileOutputStream("myfile.txt"); // Noncompliant
	// ...
	}
	} catch (Exception e) {
	// ...
	} finally {
	stream.close(); // Multiple streams were opened. Only the last is closed.
	}
	}

Los recursos deben ser cerrados	

	Correcto:
	private void readTheFile(String fileName) throws IOException {
	Path path = Paths.get(fileName);
	try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
	reader.readLine();
	// ...
	}
	// ..
	try (Stream<String> input = Files.lines("input.txt")) {
	input.forEach(System.out::println);
	}
	}
	
	private void doSomething() {
	OutputStream stream = null;
	try {
	stream = new FileOutputStream("myfile.txt");
	for (String property : propertyList) {
	// ...
	}
	} catch (Exception e) {
	// ...
	} finally {
	stream.close();
	}
	}

Los recursos deben ser cerrados	

Excepciones admitidas	Las instancias de las siguientes clases son ignoradas por esta regla porque close no tiene efecto:
	java.io.ByteArrayOutputStream
	• java.io.ByteArrayInputStream
	• java.io.CharArrayReader
	• java.io.CharArrayWriter
	• java.io.StringReader
	• java.io.StringWriter
	Java 7 introdujo la sentencia try-with-resources, que cierra implícitamente los objetos Closeable. Todos los recursos abiertos en una sentencia try-with-resources son ignorados por esta regla.
	try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
	//...
	}
	catch (...) {
	//...
	}

Clasificación	Bug

Se debe usar lógica de cortocircuito en contextos booleanos	
Problema	El uso de lógica no de cortocircuito en un contexto booleano probablemente sea un error, uno que podría causar errores graves en el programa, ya que las condiciones se evalúan bajo circunstancias incorrectas.

Corrección	Erróneo:
	if(getTrue() | getFalse()) { ... } // Noncompliant; both sides evaluated
	
	Correcto:
	if(getTrue() || getFalse()) { ... } // true short-circuit logic

Excepciones admitidas	Ninguna
Clasificación	Code smell

No se deben realizar operaciones bit a bit innecesarias	
Problema	Ciertas operaciones bit a bit son innecesarias y no deben realizarse porque sus resultados son predecibles. Específicamente, usar & -1 con cualquier valor siempre dará como resultado el valor original, al igual que cualquierValor ^ 0 y cualquierValor | 0.
Corrección	Sin información
Excepciones admitidas	Ninguna
Clasificación	Code smell

Los casos de un switch deben terminar con una declaración 'break' incondicional	
Problema	Cuando la ejecución no se termina explícitamente al final de un caso en un switch, continúa ejecutando las declaraciones del siguiente caso. Aunque esto a veces es intencional, a menudo es un error que conduce a un comportamiento inesperado.

Los casos de un switch deben terminar con una declaración 'break' incondicional	

Corrección	Erróneo:
	switch (myVariable) {
	case 1:
	foo();
	break;
	case 2: // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ?
	doSomething();
	default:
	doSomethingElse();
	break;
	}
	
	Correcto:
	switch (myVariable) {
	case 1:
	foo();
	break;
	case 2:
	doSomething();
	break;
	default:
	doSomethingElse();
	break;
	}

Los casos de un switch deben terminar con una declaración 'break' incondicional	

Excepciones admitidas	switch (myVariable) {
	case 0: // Empty case used to specify the same behavior for a group of cases.
	case 1:
	doSomething();
	break;
	case 2: // Use of a fallthrough comment
	// fallthrough
	case 3: // Use of return statement
	return;
	case 4: // Use of throw statement
	throw new IllegalStateException();
	case 5: // Use of continue statement
	continue;
	default: // For the last case, use of break statement is optional
	doSomethingElse();
	}

Clasificación	Code smell

Los TestCases deben contener pruebas	
Problema	No tiene sentido tener un JUnit TestCase sin métodos de prueba. De manera similar, no deberías tener un archivo en el directorio de pruebas llamado *Test, *Tests o *TestCase, pero sin pruebas en el archivo. Hacer cualquiera de estas cosas podría llevar a alguien a pensar que las clases no cubiertas han sido probadas. Esta regla plantea un problema cuando los archivos en el directorio de pruebas se nombran *Test, *Tests o *TestCase o implementan TestCase pero no contienen pruebas. Frameworks soportados: • JUnit3 • JUnit4 • JUnit5 • TestNG • Zohhak • ArchUnit • Pact
Corrección	Sin información
Excepciones admitidas	Ninguna
Clasificación	Code smell

Las pruebas deben incluir aserciones	
Problema	Un caso de prueba sin aserciones asegura solo que no se lancen excepciones. Más allá de la simple ejecutabilidad, no garantiza nada sobre el comportamiento del código bajo prueba. Esta regla genera una excepción cuando no se encuentran aserciones de ninguno de los siguientes frameworks conocidos en una prueba: AssertJ • Awaitility • EasyMock • Eclipse Vert.x • Fest 1.x and 2.x • Hamcrest • JMock • JMockit • JUnit • Mockito • Rest-assured 2.x, 3.x and 4.x • RxJava 1.x and 2.x • Selenide • Spring's org.springframework.test.web.servlet.ResultActions.andExpect() y org.springframework.test.web.servlet.ResultActions.andExpectAll() • Truth Framework • WireMock Además, dado que se pueden usar frameworks de aserción nuevos o personalizados, la regla se puede parametrizar para definir métodos específicos que también se considerarán como aserciones. No se generará ningún problema cuando se encuentren tales métodos en los casos de prueba. El valor del parámetro debe tener el siguiente formato <FullyQualifiedClassName>#<MethodName>, donde MethodName puede terminar con el carácter comodín. Para los constructores, el patrón debe ser <FullyQualifiedClassName>#<init>. Ejemplo: com.company.CompareToTester#compare*,com.company.CustomAssert#customAssertMethod,com.company.CheckVerifier#<init>.

Las pruebas deben incluir aserciones	

Corrección	Erróneo:
	@Test
	public void testDoSomething() { // Noncompliant
	MyClass myClass = new MyClass();
	myClass.doSomething();
	}
	
	Correcto:
	Ejemplo cuando com.company.CompareToTester#compare se utiliza como parámetro para la regla.
	import com.company.CompareToTester;
	
	@Test
	public void testDoSomething() {
	MyClass myClass = new MyClass();
	assertNull(myClass.doSomething()); // JUnit assertion
	assertThat(myClass.doSomething()).isNull(); // Fest assertion
	}
	
	@Test
	public void testDoSomethingElse() {
	MyClass myClass = new MyClass();
	new CompareToTester().compareWith(myClass); // Compliant - custom assertion method defined as rule parameter
	CompareToTester.compareStatic(myClass); // Compliant
	}

Excepciones admitidas	Ninguna
Clasificación	Code smell

Los analizadores XML no deben permitir la inclusión de archivos arbitrarios	
Problema	El estándar XML permite la inclusión de archivos XML con el elemento xinclude. Los procesadores XML reemplazarán un elemento xinclude con el contenido del archivo ubicado en la URI definida en el atributo href, potencialmente desde un almacenamiento externo como el sistema de archivos o la red, lo que puede llevar, si no se implementan restricciones, a divulgaciones de archivos arbitrarios o vulnerabilidades de falsificación de solicitudes del lado del servidor (SSRF).

Los analizadores XML no deben permitir la inclusión de archivos arbitrarios	

Corrección	Erróneo:
	Para DocumentBuilder, SAXParser, XMLInput, Transformer y Schema JAPX:
	factory.setXIncludeAware(true); // Noncompliant
	// or
	factory.setFeature("http://apache.org/xml/features/xinclude", true); // Noncompliant
	
	Librería Dom4j:
	SAXReader xmlReader = new SAXReader();
	xmlReader.setFeature("http://apache.org/xml/features/xinclude", true); // Noncompliant
	
	Librería Jdom2:
	SAXBuilder builder = new SAXBuilder();
	builder.setFeature("http://apache.org/xml/features/xinclude", true); // Noncompliant
	
	Correcto:
	Xinclude está deshabilitado por defecto y puede ser deshabilitado explícitamente como se muestra a continuación.
	Para DocumentBuilder, SAXParser, XMLInput, Transformer y Schema JAPX:
	factory.setXIncludeAware(false);
	// or
	factory.setFeature("http://apache.org/xml/features/xinclude", false);
	
	Librería Dom4j:
	SAXReader xmlReader = new SAXReader();
	xmlReader.setFeature("http://apache.org/xml/features/xinclude", false);
	
	Librería Jdom2:
	SAXBuilder builder = new SAXBuilder();
	builder.setFeature("http://apache.org/xml/features/xinclude", false);

Los analizadores XML no deben permitir la inclusión de archivos arbitrarios	

Excepciones admitidas	Esta regla no genera problemas cuando Xinclude está habilitado con un EntityResolver personalizado:
	DocumentBuilderFactory:
	DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
	factory.setXIncludeAware(true);
	// ...
	DocumentBuilder builder = factory.newDocumentBuilder();
	builder.setEntityResolver((publicId, systemId) -> new MySafeEntityResolver(publicId, systemId));
	SAXBuilder:
	SAXBuilder builder = new SAXBuilder();
	builder.setFeature("http://apache.org/xml/features/xinclude", true);
	builder.setEntityResolver((publicId, systemId) -> new MySafeEntityResolver(publicId, systemId));
	
	SAXReader:
	SAXReader xmlReader = new SAXReader();
	xmlReader.setFeature("http://apache.org/xml/features/xinclude", true);
	xmlReader.setEntityResolver((publicId, systemId) -> new MySafeEntityResolver(publicId, systemId));
	
	XMLInputFactory:
	XMLInputFactory factory = XMLInputFactory.newInstance();
	factory.setProperty("http://apache.org/xml/features/xinclude", true);
	factory.setXMLResolver(new MySafeEntityResolver());

Clasificación	Vulnerabilidad

Los analizadores XML no deberían ser vulnerables a ataques XXE	
Problema	El estándar XML permite el uso de entidades, declaradas en el DOCTYPE del documento, que pueden ser internas o externas. Al analizar el archivo XML, el contenido de las entidades externas se obtiene de un almacenamiento externo, como el sistema de archivos o la red, lo que puede llevar, si no se aplican restricciones, a divulgaciones arbitrarias de archivos o vulnerabilidades de falsificación de solicitudes del lado del servidor (SSRF). Se recomienda limitar la resolución de entidades externas utilizando una de las siguientes soluciones: • Si el DOCTYPE no es necesario, deshabilitar completamente todas las declaraciones de DOCTYPE. • Si las entidades externas no son necesarias, deshabilitar completamente sus declaraciones. • Si las entidades externas son necesarias, entonces: ◦ Utilice las características del procesador XML, si están disponibles, para autorizar solo los protocolos requeridos (por ejemplo: https). ◦ Y use un resolutor de entidades (y opcionalmente un Catálogo XML) para resolver solo las entidades de confianza.

Corrección	Erróneo:
	
	DocumentBuilder, SAXParser, XMLInput, Transformer and Schema JAPX:
	
	DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance(); // Noncompliant
	
	SAXParserFactory factory = SAXParserFactory.newInstance(); // Noncompliant
	
	XMLInputFactory factory = XMLInputFactory.newInstance(); // Noncompliant
	
	TransformerFactory factory = javax.xml.transform.TransformerFactory.newInstance(); // Noncompliant
	
	SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI); // Noncompliant
	
	Librería Dom4j:
	
	SAXReader xmlReader = new SAXReader(); // Noncompliant

Los analizadores XML no deberían ser vulnerables a ataques XXE	

	Librería Jdom2:
	SAXBuilder builder = new SAXBuilder(); // Noncompliant
	
	Correcto:
	DocumentBuilder, SAXParser, XMLInput, Transformer and Schema JAPX:
	DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
	// to be compliant, completely disable DOCTYPE declaration:
	factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
	// or completely disable external entities declarations:
	factory.setFeature("http://xml.org/sax/features/external-general-entities", false);
	factory.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
	// or prohibit the use of all protocols by external entities:
	factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
	factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
	// or disable entity expansion but keep in mind that this doesn't prevent fetching external entities
	// and this solution is not correct for OpenJDK < 13 due to a bug: https://bugs.openjdk.java.net/browse/JDK-8206132
	factory.setExpandEntityReferences(false);
	
	SAXParserFactory factory = SAXParserFactory.newInstance();
	// to be compliant, completely disable DOCTYPE declaration:
	factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
	// or completely disable external entities declarations:
	factory.setFeature("http://xml.org/sax/features/external-general-entities", false);
	factory.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
	// or prohibit the use of all protocols by external entities:
	SAXParser parser = factory.newSAXParser(); // Noncompliant
	parser.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");

Los analizadores XML no deberían ser vulnerables a ataques XXE	

	parser.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
	
	XMLInputFactory factory = XMLInputFactory.newInstance();
	// to be compliant, completely disable DOCTYPE declaration:
	factory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
	// or completely disable external entities declarations:
	factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, Boolean.FALSE);
	// or prohibit the use of all protocols by external entities:
	factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
	factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
	
	TransformerFactory factory = javax.xml.transform.TransformerFactory.newInstance();
	// to be compliant, prohibit the use of all protocols by external entities:
	factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
	factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
	
	SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
	// to be compliant, completely disable DOCTYPE declaration:
	factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
	// or prohibit the use of all protocols by external entities:
	factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
	factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
	
	Librería Dom4j:
	SAXReader xmlReader = new SAXReader();
	xmlReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
	
	Librería Jdom2:
	SAXBuilder builder = new SAXBuilder();
	builder.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");

Los analizadores XML no deberían ser vulnerables a ataques XXE	
Excepciones admitidas	Ninguna
Clasificación	Vulnerabilidad