**DIS-017 Modelo de Operación de Verificación Post-Producción**

**Anexo Reporte de Evaluación de Solución Tecnológica – Kioscos INFONAVIT (Kioskos)**

**15/oct/2018**

**Contenido**

[1. Objetivo 3](#_Toc519840867)

[2. Introducción 3](#_Toc519840868)

[3. Alcance 4](#_Toc519840869)

[4. Detalle del Entregable 5](#_Toc519840870)

[4.1. Contexto de la Solución Tecnológica 5](#_Toc519840871)

[4.2. Tipo de Solicitud 5](#_Toc519840872)

[4.3. Evaluación de Código Fuente 5](#_Toc519840873)

[4.4. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas 11](#_Toc519840874)

[4.5. Consistencia entre Ambientes del Ciclo de Vida de Desarrollo 11](#_Toc519840875)

[4.6. Cumplimiento Documental para Liberación en Ambiente Productivo 12](#_Toc519840876)

[4.7. Resultado Integral 13](#_Toc519840877)

[4.8. Propuestas de Acciones de Mejora 18](#_Toc519840878)

[5. Conclusiones 19](#_Toc519840879)

1. Objetivo

El entregable tiene por objetivo especificar las brechas y riesgos identificados derivado de la evaluación realizada sobre la solución tecnológica en los siguientes ámbitos:

**Calidad de código. -** Registrar el resultado del análisis de calidad del código fuente de una solución tecnológica específica con base en diversos estándares aceptados en la industria. La evaluación permite identificar el uso de malas prácticas, brechas de seguridad, *Bugs* que pueden provocar incidencias en la operación, deuda técnica excesiva y oportunidades de mejora en la codificación.

Una vez obtenida la información es posible generar propuestas de mejora que permitan elevar la calidad del código, reduciendo la posibilidad de afectación a la operación del Instituto y facilitando su mantenimiento.

**Verificar alineación entre diseño y construcción de la solución. -** Verificar que exista alineación entre los elementos considerados en la etapa de diseño detallado de la solución tecnológica (Capas de desarrollo, Paquetes, Clases, Métodos, Distribución de Componentes, Tecnologías) y su implementación durante la etapa construcción.

**Validar consistencia entre ambientes del ciclo de vida de desarrollo. -** Validar la consistencia entre los ambientes de desarrollo, pruebas y producción en términos de tecnologías (versiones, *releases* y parches) utilizadas.

**Documentación para liberación en ambiente productivo. -** Verificar que el proyecto cumple con la documentación necesaria para realizar la liberación de la solución tecnológica en ambiente productivo.

1. Introducción

Actualmente existen diversas problemáticas que dificultan la mejora continua de las soluciones tecnológicas desarrolladas dentro del Instituto, las cuales se mencionan a continuación:

* No se cuenta con un repositorio central de código fuente.
* Dificultad en la gestión de cambios realizados al código fuente durante el ciclo de vida de desarrollo.
* Falta de definición de estándares y mejores prácticas que regulen la construcción de soluciones.
* No se realiza la validación de la documentación necesaria para liberar en producción.
* Existen inconsistencias entre el diseño y la construcción de soluciones tecnológicas.
* Se presentan inconsistencias entre los ambientes utilizados durante el ciclo de vida de desarrollo.
* No se evalúa la calidad del código fuente generada.
* El código puesto en producción no cumple con estándares.

El equipo de verificación post-producción es responsable de cubrir las áreas de oportunidad antes mencionadas y proveer al Instituto la siguiente propuesta de valor:

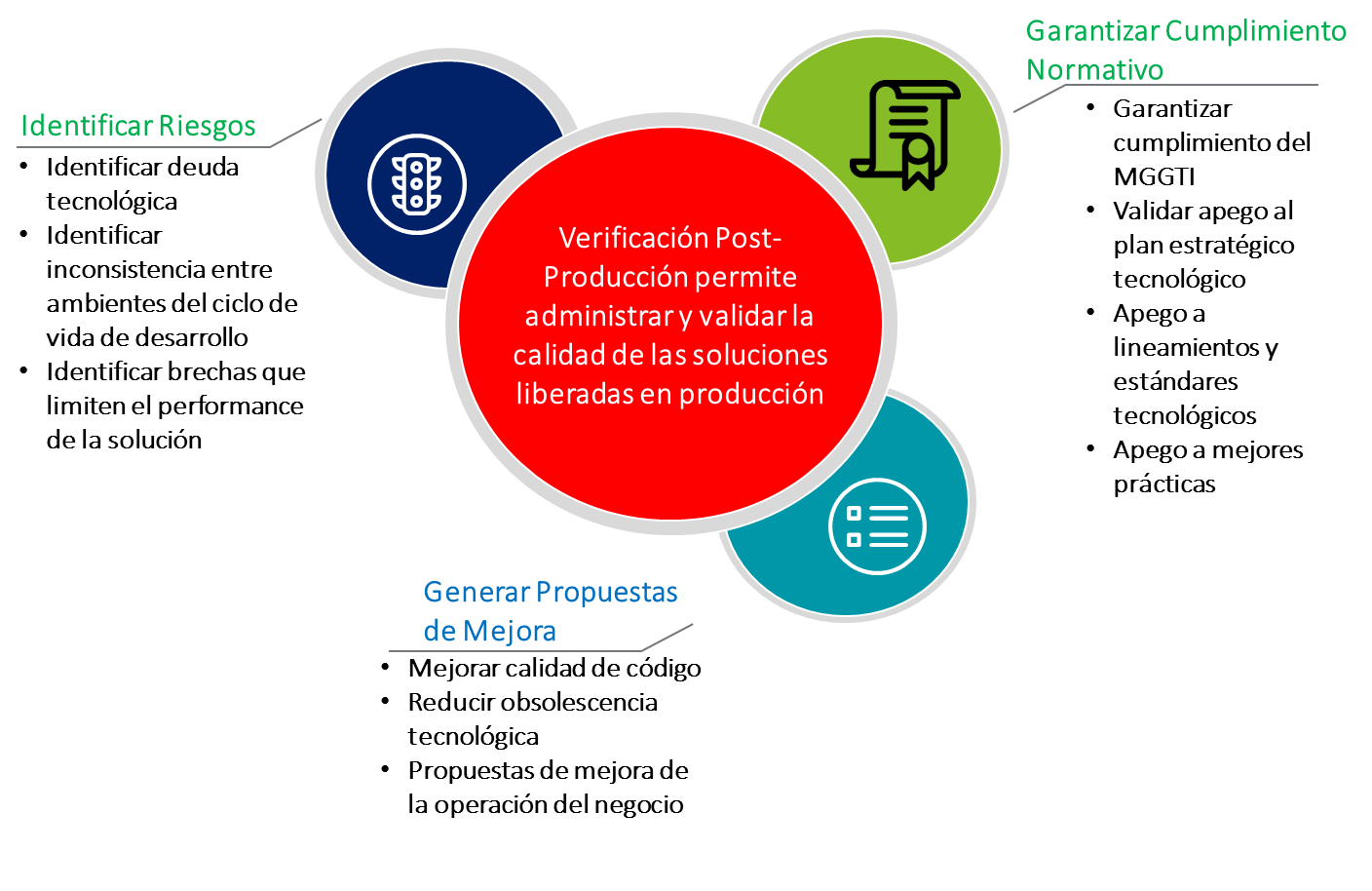


Ilustración 1 Propuesta de Valor

1. Alcance

La evaluación de soluciones tecnológicas considera los siguientes aspectos:

**Calidad de Código**: La evaluación de calidad de código se realiza con apoyo de la herramienta *SonarQube* ya que permite ejecutar una inspección estática sobre la codificación de la solución con base en los siguientes estándares*: Common Weakness Enumeration (CWE), Escal Institute of Advanced Technologies (SANS), Open Web Application Security Project (OWASP), The Motor Industry Software Reliability Association (MISRA) y Code Signing Certifcates CERTM*. La inspección permite identificar diversos errores alojados en el código que aumentan el riesgo de presentar un error en la operación.

**Alineación Entre Diseño y Construcción de Soluciones Tecnológicas:** Verificar el nivel de alineación que existe entre el diseño de soluciones tecnológicas y su construcción. De esta forma se puede identificar discrepancias entre la propuesta inicial de diseño y las decisiones consideradas en la etapa de construcción.

**Consistencia entre Ambientes del Ciclo de Vida de Desarrollo:** Validar que existe consistencia en la configuración de los ambientes (Versionamiento, *releases* y parches) utilizados a lo largo del ciclo de vida de desarrollo (Desarrollo, pruebas y producción). La validación se enfoca en los productos de *Software*, versionamiento y parches instalados en cada ambiente.

**Cumplimiento Documental para Liberación en Ambiente Productivo:** Verificar que el proyecto cumple con la documentación necesaria para realizar la liberación de la solución tecnológica en ambiente productivo.

1. Detalle del Entregable
   1. Contexto de la Solución Tecnológica

El **Sistema de Kioskos Infonavit (Kioskos)** tiene por objetivo brindar información a acreditados y derechohabientes

* 1. Tipo de Solicitud

| **Tipo de Solicitud** | Descripción | **Tipo** |
| --- | --- | --- |
| Nuevo | Construcción de una solución tecnológica nueva. |  |
| Mantenimiento Correctivo | Modificación reactiva de una solución tecnológica derivada del descubrimiento de incidencias durante la operación.  Ej. Incidencias reportadas por usuarios. |  |
| Mantenimiento Adaptivo | Modificación realizada para adaptar la solución tecnológica a su entorno.  Ej. Derivado de la actualización de versión de base de datos. |  |
| Mantenimiento Perfectivo | Modificación realizada con el objetivo de agregar funcionalidad nueva, mejorar su desempeño y/o facilitar su mantenimiento.  Ej. Requerimientos que agreguen funcionalidad. |  |
| Mantenimiento Preventivo | Modificación realizada para evitar fallas en la operación.  Ej. Optimización de Código. |  |
| Identificación de Código Fuente | Identificación de código fuente dentro del Instituto. | X |

* **Descripción de la Solicitud**

Se identificó el código fuente del proyecto como parte del esfuerzo realizado por la gerencia de Ingeniería de TI enfocado en centralizar los códigos fuente del INFONAVIT por medio de su almacenamiento en el repositorio institucional.

* 1. Evaluación de Código Fuente

La inspección permite identificar diversas incidencias alojados en el código los cuales aumentan el riesgo de presentar una falla en la operación. La herramienta de evaluación de calidad de código *SonarQube* agrupa las incidencias identificadas por medio de categorías que a su vez se clasifican por su severidad.

* **Categoría**
  + ***Bugs***

Los *Bugs* son errores en el código fuente de un sistema o aplicación que deriva en un comportamiento incorrecto/inesperado generando un resultado indeseado. Es importante identificar y solucionar de manera temprana los *Bugs* de programación para asegurar la calidad en los productos desarrollados, así como su funcionamiento de acuerdo a lo esperado.

* + **Vulnerabilidades de Seguridad**

Las Vulnerabilidades de Seguridad son imperfecciones alojadas en el código fuente las cuales son susceptibles a ser aprovechadas por una atacante con el fin ejecutar código malicioso y/o acceder a información sensible manipulada por una solución tecnológica.

Por lo anterior, es importante identificar de manera temprana las vulnerabilidades de seguridad para así fortalecer el código fuente y, en consecuencia, generar productos de *software* seguros que disminuyan el riesgo de ataques por parte de terceros.

* + **Optimizadores**

Los optimizadores o “*Code* *Smells*” representan un indicador de un problema más profundo. Este tipo de observaciones **NO** representan un *Bug* o falla técnica dentro del código, señalan diversos factores que pueden causar deficiencias a futuro: Aumento de presencia de fallas, bajo desempeño e incluso aumento de deuda técnica.

Por lo anterior, es importante identificar de manera temprana los optimizadores necesarios para aplicar medidas correctivas, de esta forma aumentaría la calidad del producto de *software* y reducir el riesgo de presencia de fallas durante la operación de la solución tecnológica.

* **Severidades**

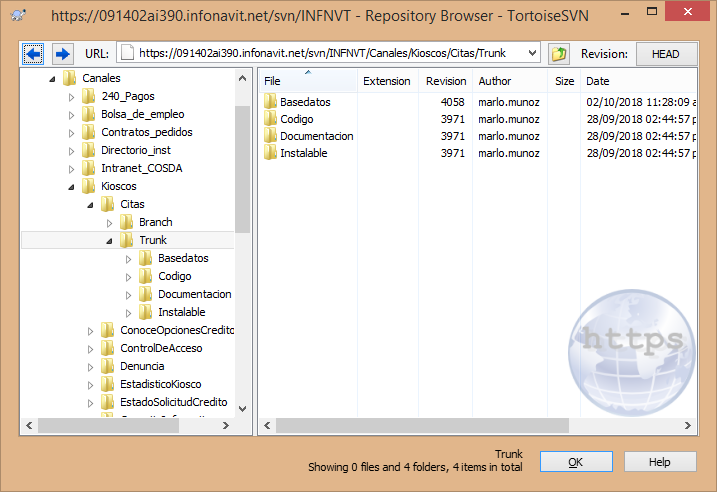
Los errores categorizados anteriormente son clasificados por severidades las cuales son definidas con base en 2 factores: **Impacto** y **Probabilidad**.

|  | **Impacto** | **Probabilidad** | **Descripción** |
| --- | --- | --- | --- |
| ***Blocker*** |  |  | * Alta probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de gran escala que imposibilite la ejecución correcta de la operación del Instituto * Su corrección debe ser inmediata |
| ***Critical*** |  |  | * Baja probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de gran escala que imposibilite la ejecución correcta de la operación del Instituto * Su corrección debe ser inmediata |
| ***Major*** |  |  | * Alta probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de afectación menor |
| ***Minor*** |  |  | * Baja probabilidad de ocurrencia * En caso de ocurrir puede provocar un error de afectación menor |

* + 1. Ubicación en el Repositorio Institucional

Los artefactos evaluados (Código fuente y/o ejecutables) se encuentran en la siguiente ubicación dentro del repositorio:

[https://091402ai390.infonavit.net/svn/INFNVT/Canales/](https://091402ai390.infonavit.net/svn/INFNVT/Canales/Portal_Mi_Cuenta)Kioscos/Citas/Trunk



El proyecto “KIOSKOS” se compone de los siguientes proyectos los cuales fueron almacenados en directorios independientes con una estructura Trunk-Branch propia:

Citas

ConoceOpcionesCredito

ControlDeAcceso

Denuncia

EstadisticoKiosco

EstadoSolicitudCredito

GarantiaInfonavit

Precalificacion

Quejas

Recompensas

RegistroMCI

SaldosyMov

* + 1. Estructura de Archivo *sonar.properties*

La inspección se realiza por medio de la definición del *archivo sonar.properties* el cual indica a la herramienta *Sonar Scanner* los parámetros necesarios para ejecutar el análisis de calidad de código.

#################################################

# Caracteres Unicode con acentos:

# ===============================

#

# a - \u00e1 A - \u00c1

# e - \u00e9 E - \u00c9

# i - \u00ed I - \u00cd

# o - \u00f3 O - \u00d3

# u - \u00fa U - \u00da

#

# Caracteres Unicode con diéresis

# ===============================

# u - \u00fc U - \u00dc

#

# Caracteres Unicode con tilde

# ===============================

#

# n - \u00f1 N - \u00d1

#

# Interrogación que abre - \u00bf

#

#

################################################

#Datos Generales del Proyecto

sonar.projectKey=mx.org.infonavit.kioskos

sonar.projectName=Kioskos

sonar.projectVersion=1.0

sonar.scm.disabled=True

#Módulos

sonar.modules=Citas,ConoceOpcionesCredito,ControlDeAcceso,Denuncia,EstadisticoKiosco,EstadoSolicitudCredito,GarantiaInfonavit,Precalificacion,Quejas,Recompensas,RegistroMCI,SaldosyMov

Citas.sonar.projectName=Citas

ConoceOpcionesCredito.sonar.projectName=ConoceOpcionesCredito

ControlDeAcceso.sonar.projectName=ControlDeAcceso

Denuncia.sonar.projectName=Denuncia

EstadisticoKiosco.sonar.projectName=EstadisticoKiosco

EstadoSolicitudCredito.sonar.projectName=EstadoSolicitudCredito

GarantiaInfonavit.sonar.projectName=GarantiaInfonavit

Precalificacion.sonar.projectName=Precalificacion

Quejas.sonar.projectName=Quejas

Recompensas.sonar.projectName=Recompensas

RegistroMCI.sonar.projectName=RegistroMCI

SaldosyMov.sonar.projectName=SaldosyMov

#Sources

Citas.sonar.sources=.

ConoceOpcionesCredito.sonar.sources=.

ControlDeAcceso.sonar.sources=.

Denuncia.sonar.sources=.

EstadisticoKiosco.sonar.sources=.

EstadoSolicitudCredito.sonar.sources=.

GarantiaInfonavit.sonar.sources=.

Precalificacion.sonar.sources=.

Quejas.sonar.sources=.

Recompensas.sonar.sources=.

RegistroMCI.sonar.sources=.

SaldosyMov.sonar.sources=.

#Bin

Citas.sonar.java.binaries=NuevoKioscos-Eflow/NvoCitas/WebContent/WEB-INF/classes

ConoceOpcionesCredito.sonar.java.binaries=NvoConoceOpcionesCredito/WebContent/WEB-INF/classes

ControlDeAcceso.sonar.java.binaries=NvoControlDeAcceso/WebContent/WEB-INF/classes

Denuncia.sonar.java.binaries=NvoDenuncia/WebContent/WEB-INF/classes

EstadisticoKiosco.sonar.java.binaries=EstadisticosKiosco/WebContent/WEB-INF/classes

EstadoSolicitudCredito.sonar.java.binaries=NvoEstadoSolicitudCredito/WebContent/WEB-INF/classes

GarantiaInfonavit.sonar.java.binaries=NvoGarantiaInfonavit/WebContent/WEB-INF/classes

Precalificacion.sonar.java.binaries=NvoPrecalificacion/WebContent/WEB-INF/classes

Quejas.sonar.java.binaries=NvoQuejas/WebContent/WEB-INF/classes

Recompensas.sonar.java.binaries=NvoRecompensas/WebContent/WEB-INF/classes

RegistroMCI.sonar.java.binaries=NvoMiCuentaInfonavit/WebContent/WEB-INF/classes

SaldosyMov.sonar.java.binaries=NvoSaldosyMovimientos/WebContent/WEB-INF/classes

#Librerías

Citas.sonar.java.libraries=NuevoKioscos-Eflow/NvoCitas/WebContent/WEB-INF/lib

ConoceOpcionesCredito.sonar.java.libraries=NvoConoceOpcionesCredito/WebContent/WEB-INF/lib

ControlDeAcceso.sonar.java.libraries=NvoControlDeAcceso/WebContent/WEB-INF/lib

Denuncia.sonar.java.libraries=NvoDenuncia/WebContent/WEB-INF/lib

EstadisticoKiosco.sonar.java.libraries=EstadisticosKiosco/WebContent/WEB-INF/lib

EstadoSolicitudCredito.sonar.java.libraries=NvoEstadoSolicitudCredito/WebContent/WEB-INF/lib

GarantiaInfonavit.sonar.java.libraries=NvoGarantiaInfonavit/WebContent/WEB-INF/lib

Precalificacion.sonar.java.libraries=NvoPrecalificacion/WebContent/WEB-INF/lib

Quejas.sonar.java.libraries=NvoQuejas/WebContent/WEB-INF/lib

Recompensas.sonar.java.libraries=NvoRecompensas/WebContent/WEB-INF/lib

RegistroMCI.sonar.java.libraries=NvoMiCuentaInfonavit/WebContent/WEB-INF/lib

SaldosyMov.sonar.java.libraries=NvoSaldosyMovimientos/WebContent/WEB-INF/lib

#Exclusions

sonar.exclusions=.classpath,.factorypath,.pmd,.project,.apt\_generated,.settings,\*\*/\*.MF,.websettings

sonar.cpd.exclusions=.classpath,.factorypath,.pmd,.project,.apt\_generated,.settings,\*\*/\*.MF,.websettings



* + 1. Ejecución de Análisis de Código Fuente

El análisis de la calidad del código fuente se realiza por medio de la ejecución del siguiente comando: ***sonar-scaner.bat***

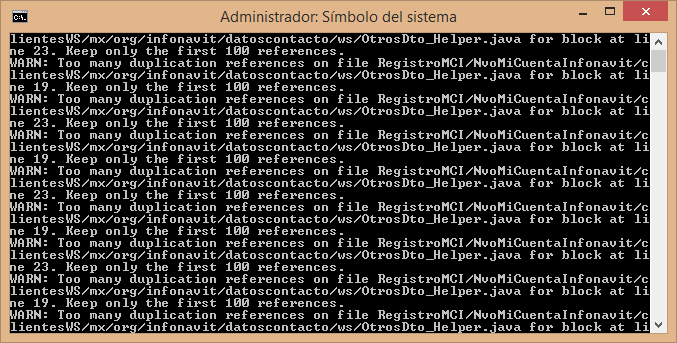


Ilustración 2 Pantalla 1 de ejecución *sonar-scaner.bat*

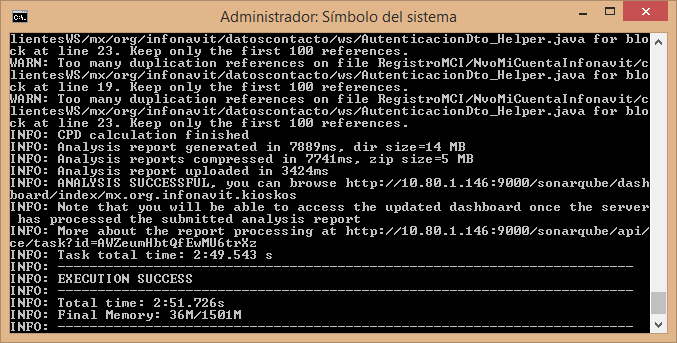


Ilustración 3 Pantalla 2 de ejecución *sonar-scaner.bat*

* + 1. Resultado del Análisis de Código Fuente

**Proyecto**: Citas

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 26 |
| *Blocker* | 1 |
| *Critical* | 2 |
| *Major* | 17 |
| *Minor* | 6 |
| Vulnerabilidades | *Total* | 17 |
| *Blocker* | 3 |
| *Critical* | 3 |
| *Major* | 11 |
| *Minor* | 0 |
| Optimizadores | *Total* | 2634 |
| *Blocker* | 4 |
| *Critical* | 306 |
| *Major* | 718 |
| *Minor* | 1028 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| Bug | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 1 |
| Bug | *Critical* | **Los resultados de "*in*" y "*instanceof*" deben ser negados en lugar de operandos**  Combinar el orden de las operaciones casi siempre producirá resultados inesperados.  Del mismo modo, la negación mal aplicada también producirá malos resultados. Por ejemplo, considere la diferencia entre !*key* in *dict* y !(clave en *dic*). El primero busca un valor booleano (!*key*) en *dic*, y el otro busca una cadena e invierte el resultado. !obj de *SomeClass* tiene el mismo problema. | 1 |
| Bug | *Critical* | **Las llamadas de función no deben pasar argumentos adicionales**  Es posible llamar a una función *JavaScript* con más argumentos de los necesita, pero los argumentos adicionales serán ignorados por la ejecución de la función. | 1 |
| Bug | *Major* | **Las sentencias de salto no deben ocurrir en bloques "*Finally*”**  Utilizar return, *break*, *throw*, etc. desde un bloque final se suprime la propagación de cualquier *throwable* no manejado que haya sido lanzado en el bloque de *try* o *catch*. | 5 |
| Bug | *Major* | **No se deben utilizar operadores de igualdad estricta con tipos diferentes**  Realizar comparaciones de tipos diferentes utilizando operadores de igualdad estricta como === y !== siempre devolverá el mismo valor ya que no se realiza ninguna conversión de tipo antes de la comparación, por lo que se deduce se trata de un bug inmerso en el código. | 3 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 3 |
| Bug | *Major* | **No se debe acceder a las propiedades de las variables con valores "*null*" o "*undefined*"**  Cuando a una variable se le asigna un valor no *undefined* o *null*, no se generan propiedades. Intentar acceder a las propiedades de una variable de este tipo resulta en un *TypeError*, causando una terminación abrupta del *script* si el error no se detecta en un bloque de captura. | 2 |
| Bug | *Major* | **No se deben hacer verificaciones de igualdad**  Las comparaciones de diferentes tipos siempre dan como resultado *false*. Por lo que las comparaciones y el código dependiente debe eliminarse. Esto incluye:   * Comparando un objeto con *null.* * Comparando un objeto con un primitivo no emparentado (E. G. *string* con un int). | 2 |
| Bug | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 1 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 1 |
| Vulnerabilidad | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 3 |
| Vulnerabilidad | *Critical* | **El código no debe ser inyectado y ejecutado dinámicamente**  La función de *eval* es una manera de ejecutar código arbitrario en tiempo de ejecución. La evaluación dinámica del código es lenta y un problema potencial de seguridad cuando los argumentos no han sido validados correctamente. | 3 |
| Vulnerabilidad | *Minor* | **No debe utilizarse "*alert(...)"***  *alert(....)* así como *confirm(....)* y *prompt(...)* puede de utilidad para depurar durante la etapa de desarrollo, pero en ambiente de producción este tipo de *pop-up* podría exponer información sensible a los atacantes, por lo que no debe mostrarse. | 7 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador. | 2 |
| Vulnerabilidad | *Minor* | **Las direcciones *IP* no deben estar en código duro**  Poner en duro dirección *IP* dentro del código fuente es una mala idea por varias razones:   * Se necesita compilar de nuevo si la dirección *IP* cambia. * Obliga a utilizar la misma dirección en todos los ambientes (Desarrollo, Pruebas y Producción). * Deja la responsabilidad de colocar el valor a utilizar en producción en manos del desarrollador. * Permite a los atacantes descompilar el código y descubrir direcciones potencialmente sensibles. | 2 |
| Optimizador | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 3 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 1 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 233 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 53 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 9 |
| Optimizador | *Critical* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 5 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 4 |
| Optimizador | *Critical* | **La ejecución del *Garbage* *Collector* sólo debe ser activada por la *JVM***  No se recomienda llamar a *System.gc()* o *Runtime.getRuntime(). gc()* debido a que no hay manera de saber exactamente lo que se hará bajo la campana por el *JVM* debido a que el comportamiento dependerá de su proveedor, versión y opciones.  Una aplicación que se basa en estos métodos impredecibles también es impredecible y, por lo tanto aumenta el riesgo de fallo. La tarea de ejecutar el recolector de basura debe dejarse exclusivamente al *JVM*. | 1 |
| Optimizador | *Critical* | **Los tipos de *wildcard* genéricos no deben utilizarse en los parámetros de *return***  Se recomienda no utilizar *wildcards* como tipos de devolución. Debido a que las reglas de inferencia de tipo son bastante complejas, es poco probable que el usuario de esa *API* sepa cómo utilizarla correctamente. El uso de tipos de *wildcards* debe limitarse a los parámetros del método. | 1 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico.   Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 147 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 104 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 96 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 89 |
| Optimizador | *Major* | **Deben eliminarse los métodos "privados" no utilizados**  Métodos privados que nunca se ejecutan son código muerto, código innecesario e inoperante que debe ser eliminado. La eliminación del código muerto disminuye el tamaño de la base de códigos mantenida, lo que facilita la comprensión del programa y evita la introducción de errores. | 67 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 52 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 35 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 34 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 28 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 22 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no | 10 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 10 |
| Optimizador | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 5 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 4 |
| Optimizador | *Major* | **Deben eliminarse los métodos "privados" no utilizados**  Métodos privados que nunca se ejecutan son código muerto, código innecesario e inoperante que debe ser eliminado. La eliminación del código muerto disminuye el tamaño de la base de códigos mantenida, lo que facilita la comprensión del programa y evita la introducción de errores. | 3 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 466 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 456 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 144 |
| Optimizador | *Minor* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 142 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 66 |
| Optimizador | *Minor* | **Los métodos de anulación deben hacer más que simplemente llamar al mismo método en la clase *super* (*super* *class*)**  Anular un método sólo para llamar al mismo método desde la superclase sin realizar ninguna otra acción es inútil y engañoso. La única vez que esto se justifica es en los métodos de anulación final, donde el efecto es bloquear el comportamiento de la clase padre. Esta regla ignora anulaciones tales como *equals*, *hashCode* y *toString.* | 61 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 43 |
| Optimizador | *Minor* | **Los bloques de código anidados no deben utilizarse**  Los bloques de código anidado se pueden utilizar para crear un nuevo alcance y re*string*ir la visibilidad de las variables definidas en su interior. El uso de esta característica en un método indica típicamente que el método tiene demasiadas responsabilidades, y debe ser refacturado en métodos más pequeños. | 43 |
| Optimizador | *Minor* | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 30 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 27 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 24 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 20 |
| Optimizador | *Minor* | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 16 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 15 |
| Optimizador | *Minor* | ***Collection.isEmpty()* debe usarse para probar vacíos**  *Collection.size ()* se usa para probar los trabajos vacíos, pero el uso de *Collection.isEmpty* () hace que el código sea más legible y más eficaz. La complejidad temporal de cualquier implementación del método *isEmpty* () debe ser *O(1)* considerando que algunas implementaciones de tamaño () pueden ser *O(n*). | 9 |

**Proyecto**: ConoceOpcionesCredito

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 0 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 0 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 0 |
| Optimizadores | *Total* | 97 |
| *Blocker* | 0 |
| *Critical* | 20 |
| *Major* | 25 |
| *Minor* | 52 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Optimizador* | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 18 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada. | 2 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 8 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 6 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico.   Mejora la legibilidad del código fuente al hacer evidente | 4 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 2 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 2 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 1 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 1 |
| Optimizador | *Major* | **Los métodos no deberían tener implementaciones idénticas**  La duplicidad en la implementación de métodos generalmente se presenta por un error o la intención explicita de duplicar la implementación. La duplicidad de implementación de métodos puede ser confusa para el equipo responsable de dar mantenimiento al código por lo que se recomienda eliminar la duplicidad e implementar la invocación de una aplicación a otra. | 1 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 15 |
| Optimizador | *Minor* | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 11 |
| Optimizador | *Minor* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 9 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 4 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 3 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 3 |
| Optimizador | *Minor* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 1 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 1 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 1 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código | 1 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 1 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 1 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 1 |

**Proyecto**: ControlDeAcceso

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 5 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 5 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 0 |
| *Blocker* | 1 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 2 |
| Optimizadores | *Total* | 884 |
| *Blocker* | 0 |
| *Critical* | 48 |
| *Major* | 225 |
| *Minor* | 611 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| Bug | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 3 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 3 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 1 |
| Vulnerabilidad | *Blocker* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 2 |
| Vulnerabilidad | *Minor* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 1 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 24 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 9 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 7 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 6 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 2 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 112 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 24 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 22 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 15 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación | 14 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 10 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 6 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 4 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 3 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 2 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 2 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 2 |
| Optimizador | *Major* | **Los bloques *Try-catch* no deben estar anidados**  Anidar *try/catch* bloques afecta gravemente a la legibilidad del código fuente porque hace demasiado difícil entender qué bloque atrapará cada excepción | 2 |
| Optimizador | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 2 |
| Optimizador | *Major* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 1 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 223 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 143 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 66 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 61 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 29 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 29 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 18 |
| Optimizador | *Minor* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 7 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 6 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 5 |
| Optimizador | *Minor* | Los literales booleanos no deberían ser redundantes  Los literales booleanos redundantes deben eliminarse de las expresiones para mejorar la legibilidad. | 4 |
| Optimizador | *Minor* | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 4 |
| Optimizador | *Minor* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 3 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 3 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 2 |

**Proyecto**: Denuncia

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 7 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 7 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 12 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 12 |
| Optimizadores | *Total* | 337 |
| *Blocker* | 0 |
| *Critical* | 36 |
| *Major* | 99 |
| *Minor* | 201 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 5 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action* | 1 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 1 |
| *Vulnerabilidad* | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 5 |
| *Vulnerabilidad* | *Minor* | **Las direcciones *IP* no deben estar en código duro**  Poner en duro dirección *IP* dentro del código fuente es una mala idea por varias razones:   * Se necesita compilar de nuevo si la dirección *IP* cambia. * Obliga a utilizar la misma dirección en todos los ambientes (Desarrollo, Pruebas y Producción). * Deja la responsabilidad de colocar el valor a utilizar en producción en manos del desarrollador. * Permite a los atacantes descompilar el código y descubrir direcciones potencialmente sensibles. | 4 |
| *Vulnerabilidad* | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador. | 2 |
| *Vulnerabilidad* | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 1 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 27 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 4 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada. | 3 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 23 |
| Optimizador | *Major* | **Los bloques *Try-catch* no deben estar anidados**  Anidar *try/catch* bloques afecta gravemente a la legibilidad del código fuente porque hace demasiado difícil entender qué bloque atrapará cada excepción | 14 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 13 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.*   *StringBuilder* en lugar de *StringBuffer.* | 10 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 8 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 5 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 5 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 4 |
| Optimizador | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 4 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 3 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 3 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 1 |
| Optimizador | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 1 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 1 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 110 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 19 |
| Optimizador | *Minor* | Las cláusulas "*catch*" deben hacer más actividades además de lanzar  Una cláusula de captura que sólo vuelve a lanzar la excepción capturada tiene el mismo efecto que omitir la captura por complete. Tales cláusulas deberían eliminarse o completarse con la lógica apropiada. | 14 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 13 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 12 |
| Optimizador | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 5 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 4 |
| Optimizador | *Minor* | Los literales booleanos no deberían ser redundantes  Los literales booleanos redundantes deben eliminarse de las expresiones para mejorar la legibilidad. | 4 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 4 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 4 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 3 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 2 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 2 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 2 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 2 |

**Proyecto**: EstadisticoKiosco

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 187 |
| *Blocker* | 31 |
| *Critical* | 3 |
| *Major* | 129 |
| *Minor* | 24 |
| Vulnerabilidades | *Total* | 29 |
| *Blocker* | 4 |
| *Critical* | 14 |
| *Major* | 0 |
| *Minor* | 11 |
| Optimizadores | *Total* | 886 |
| *Blocker* | 59 |
| *Critical* | 87 |
| *Major* | 293 |
| *Minor* | 447 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 31 |
| Bug | *Critical* | **Las llamadas de función no deben pasar argumentos adicionales**  Es posible llamar a una función *JavaScript* con más argumentos de los necesita, pero los argumentos adicionales serán ignorados por la ejecución de la función. | 2 |
| Bug | *Critical* | **Los resultados de "*in*" y "*instanceof*" deben ser negados en lugar de operandos**  Combinar el orden de las operaciones casi siempre producirá resultados inesperados.  Del mismo modo, la negación mal aplicada también producirá malos resultados. Por ejemplo, considere la diferencia entre !*key* in *dict* y !(clave en *dic*). El primero busca un valor booleano (!*key*) en *dic*, y el otro busca una cadena e invierte el resultado. !obj de *SomeClass* tiene el mismo problema. | 1 |
| *Bug* | *Major* | **Las sentencias de salto no deben ocurrir en bloques "*Finally*”**  Utilizar return, *break*, *throw*, etc. desde un bloque final se suprime la propagación de cualquier *throwable* no manejado que haya sido lanzado en el bloque de *try* o *catch*. | 46 |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 46 |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 14 |
| *Bug* | *Major* | **No se deben hacer verificaciones de igualdad**  Las comparaciones de diferentes tipos siempre dan como resultado *false*. Por lo que las comparaciones y el código dependiente debe eliminarse. Esto incluye:   * Comparando un objeto con *null.* * Comparando un objeto con un primitivo no emparentado (E. G. *string* con un int). * Comparación de clases no relacionadas. * Comparación de una clase e interfaz no relacionada. * Comparación de tipos de interfaz no relacionados. * Comparando un *array* con un *no-array*. * Comparando dos *arrays.*   Específicamente en el caso de los *arrays*, dado que los *arrays* no anulan *Objects.equals(),* llamar iguales en dos *arrays* es lo mismo que comparar sus direcciones. Esto significa que *array1*.*equals*(*array2)* es equivalente a *array1*===*array2*.  Sin embargo, algunos equipos de desarrollo pueden esperar que *Array.equals(Object obj*) haga más que una simple comparación de direcciones de memoria, comparando por ejemplo el tamaño y contenido de los dos *arrays*. En su lugar, == *operator* o *Arrays.equals(array1, array2*) siempre debe ser usado con *arrays*. | 6 |
| *Bug* | *Major* | **No se debe acceder a las propiedades de las variables con valores "*null*" o "*undefined*"**  Cuando a una variable se le asigna un valor no *undefined* o *null*, no se generan propiedades. Intentar acceder a las propiedades de una variable de este tipo resulta en un *TypeError*, causando una terminación abrupta del *script* si el error no se detecta en un bloque de captura. | 5 |
| *Bug* | *Major* | **No se deben utilizar operadores de igualdad estricta con tipos diferentes**  Realizar comparaciones de tipos diferentes utilizando operadores de igualdad estricta como === y !== siempre devolverá el mismo valor ya que no se realiza ninguna conversión de tipo antes de la comparación, por lo que se deduce se trata de un bug inmerso en el código. | 3 |
| *Bug* | *Major* | **Los campos *Non-thread-safe* no deben ser estáticos**  No todas las clases de la librería *Java* estándar están escritas para ser seguras con los hilos. Es muy probable que su uso de múltiples subprocesos cause problemas con los datos o excepciones en tiempo de ejecución.  Esta regla plantea un problema cuando una instancia de validación de Calendar, *DateFormat, Javax. xml. xpath. XPath, orJavax. xml. SchemaFactory* está marcada como estática | 3 |
| *Bug* | *Major* | **Las sentencias de salto no deben ocurrir en bloques "*Finally*”**  Utilizar return, *break*, *throw*, etc. desde un bloque final se suprime la propagación de cualquier *throwable* no manejado que haya sido lanzado en el bloque de *try* o *catch*. | 2 |
| *Bug* | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 2 |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 1 |
| *Bug* | *Major* | **Las sentencias de salto no deben ir seguidas de otras sentencias**  Las sentencias *Jump* (*return*, *break* y *continue*) y expresiones *throw* mueven el flujo de control fuera del bloque de código actual. | 1 |
| Bug | *Minor* | **El *boxing* y *unboxing* no debe revertirse inmediatamente**  *Boxing* es el proceso de poner un valor primitivo en un objeto análogo, como por ejemplo crear un entero para mantener un valor. El *Unboxing* es el proceso de recuperar el valor primitivo de tal objeto.    Dado que el valor original no se modifica durante el *boxing* y el *unboxing*, no tiene sentido hacerlo cuando no es necesario. Esto también se aplica al *autoboxing* y *auto*-*unboxing.* | 12 |
| Bug | *Minor* | **Los parámetros del método, las excepciones capturadas y los valores iniciales de las variables *foreach* no deben ignorarse**  Técnicamente correcto asignar a los parámetros desde dentro de los cuerpos del método, hacerlo antes de que se lea el valor del parámetro es considerado un error. En cambio, los valores iniciales de los parámetros, las excepciones capturadas y los parámetros de *foreach* deben ser tratados como finales o leídos antes de la reasignación. | 10 |
| Bug | *Minor* | **No deben utilizarse sentencias “with”**  El uso de la palabra clave “with” produce un error en el código JavaScript.  Usar “with” permite un acceso rápido a las propiedades de un objeto, asumiendo que ya están configuradas. Pero úsalo para acceder a alguna propiedad que no esté ya establecida en el objeto. Dado que los efectos de “with” dependen completamente del objeto que se le ha pasado, “with” puede ser peligrosamente impredecible, y nunca debe ser utilizado. | 1 |
| Bug | *Minor* | **Los operandos matemáticos deben unirse antes de la asignación**  Cuando una operación aritmética se realiza en números enteros, el resultado siempre será un número entero. Puede asignar ese resultado a un valor *long*, *double* o *float* con una conversión automática de tipo de dato, pero en caso de haber comenzado como *int* o *long*, es probable que el resultado no sea el esperado.  Por ejemplo, si el resultado de la división *int* se asigna a una variable de punto flotante, la precisión se habrá perdido antes de la asignación.  Para evitar resultados imprevistos por lo menos un operando debe ser lanzado o promovido al tipo de dato final antes de que la operación tenga lugar. | 1 |
| Vulnerabilidad | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 3 |
| Vulnerabilidad | *Blocker* | **Deben utilizarse mecanismos de vinculación *Structured Query Language* (*SQL*)**  Las aplicaciones que ejecutan comandos *SQL* deben neutralizar cualquier valor proporcionado externamente utilizado en esos comandos. De lo contrario, un atacante podría incluir cadenas para modificar la consulta de manera que se ejecuten comandos involuntarios o se expongan datos confidenciales.  Esta regla verifica una variedad de métodos de diferentes *frameworks* que son susceptibles a la inyección *SQL* si no se usan correctamente. Los *frameworks* que se cubren son Java *Java Data Base Connection*, *Java Persistence API (JPA)*, *Java Data Objects (JDO),* *Hibernate* y *Spring*. Se prueban los siguientes métodos específicos:   * *org.hibernate.Session.createQuery* * *org.hibernate.Session.createSQLQuery* * *java.sql.Statement.executeQuery* * *java.sql.Statement.execute* * *java.sql.Statement.executeUpdate* * *java.sql.Statement.executeLargeUpdate* * *java.sql.Statement.addBatch* * *java.sql.Connection.prepareStatement* * *java.sql.Connection.prepareCall* * *java.sql.Connection.nativeSQL* * *javax.persistence.EntityManager.createNativeQuery* * *javax.persistence.EntityManager.createQuery* * *org.springframework.jdbc.core.JdbcOperations.batchUpdate* * *org.springframework.jdbc.core.JdbcOperations.execute* * *org.springframework.jdbc.core.JdbcOperations.query* * *org.springframework.jdbc.core.JdbcOperations.queryForList* * *org.springframework.jdbc.core.JdbcOperations.queryForMap* * *org.springframework.jdbc.core.JdbcOperations.queryForObject* * *org.springframework.jdbc.core.JdbcOperations.queryForRowSet* * *org.springframework.jdbc.core.JdbcOperations.queryForInt* * *org.springframework.jdbc.core.JdbcOperations.queryForLong* * *org.springframework.jdbc.core.JdbcOperations.update* * *org.springframework.jdbc.core.PreparedStatementCreatorFactory.<init>* * *org.springframework.jdbc.core.PreparedStatementCreatorFactory.newPreparedStatementCreator* * *javax.jdo.PersistenceManager.newQuery* * *javax.jdo.Query.setFilter* * *javax.jdo.Query.setGrouping*   Si se define un método en una interfaz, también se prueban las implementaciones. Por ejemplo, este es el caso de *org.springframework.jdbc.core.JdbcOperations*, que usualmente es usado como org*.springframework.jdbc.core.JdbcTemplate* | 1 |
| Vulnerabilidad | *Critical* | **El código no debe ser inyectado y ejecutado dinámicamente**  La función de *eval* es una manera de ejecutar código arbitrario en tiempo de ejecución. La evaluación dinámica del código es lenta y un problema potencial de seguridad cuando los argumentos no han sido validados correctamente. | 11 |
| Vulnerabilidad | *Critical* | **No se deben utilizar constructores de funciones**  Los constructores de funciones aumentan el riesgo de fallo ya que su ejecución evalúa los argumentos de cadena del constructor de forma similar a la forma en que funciona la evaluación, lo que podría exponer la aplicación programa a un código aleatorio no intencionado que puede ser lento y un riesgo para la seguridad.  En general, es mejor evitarlo por completo, especialmente cuando se utiliza para analizar datos *JSON*. Se sugiere utilizar las funciones *JSON* incorporadas de *ECMAScript* 5 o una biblioteca dedicada. | 3 |
| Vulnerabilidad | *Minor* | **No debe utilizarse "*alert(...)"***  *alert(....)* así como *confirm(....)* y *prompt(...)* puede de utilidad para depurar durante la etapa de desarrollo, pero en ambiente de producción este tipo de *pop-up* podría exponer información sensible a los atacantes, por lo que no debe mostrarse. | 10 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 1 |
| Optimizador | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 59 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 78 |
| Optimizador | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 3 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 2 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 2 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 1 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 1 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 98 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 50 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 25 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 24 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 23 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 20 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 10 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 6 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 6 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 5 |
| Optimizador | *Major* | **Las funciones no deben definirse dentro de los bucles**  Definir una función dentro de un bucle puede producir resultados inesperados. Esta función mantiene referencias a las variables que se definen en los alcances externos. Todas las instancias de función creadas dentro del bucle, por lo tanto, ven los mismos valores para estas variables, lo que probablemente no se espera. | 4 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 4 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 3 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 3 |
| Optimizador | *Major* | **Los métodos no deberían tener implementaciones idénticas**  La duplicidad en la implementación de métodos generalmente se presenta por un error o la intención explicita de duplicar la implementación. La duplicidad de implementación de métodos puede ser confusa para el equipo responsable de dar mantenimiento al código por lo que se recomienda eliminar la duplicidad e implementar la invocación de una aplicación a otra. | 3 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 86 |
| Optimizador | *Minor* | Las declaraciones deben utilizar interfaces de *Java* *Collection* como "*List*" en lugar de clases de implementación específicas como "*LinkedList*"  El propósito de *Java* *Collections* *API* es proporcionar una jerarquía bien definida de interfaces para ocultar los detalles de implementación.  Las clases de implementación deben ser utilizadas para instanciar nuevas colecciones, pero el resultado de una instanciación debe ser almacenado idealmente en una variable cuyo tipo sea una interfaz de colección *Java*.  Esta regla plantea un problema cuando una clase de implementación:   * Es devuelto de un método público. * Se acepta como argumento para un método público. * Se expone como miembro público. | 57 |
| Optimizador | *Minor* | ***Strings* no deben concatenarse usando' +' en un bucle**  Los *strings* son objetos inmutables, por lo que la concatenación no se limita a añadir las nuevas cadenas al final del *string* existente. En su lugar, en cada iteración de bucle, el primer *String* debe convertirse en un tipo de objeto intermedio, añadir el segundo *string* y a continuación, el objeto intermedio debe convertirse de nuevo en un *String*. El rendimiento de estas operaciones intermedias se degrada a medida que el *String* se alarga, por lo tanto, se sugiere el uso de *StringBuilder.* | 48 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 43 |
| Optimizador | *Minor* | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 36 |
| Optimizador | *Minor* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 32 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 32 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 27 |
| Optimizador | *Minor* | ***Collection.isEmpty()* debe usarse para probar vacíos**  *Collection.size ()* se usa para probar los trabajos vacíos, pero el uso de *Collection.isEmpty* () hace que el código sea más legible y más eficaz. La complejidad temporal de cualquier implementación del método *isEmpty* () debe ser *O(1)* considerando que algunas implementaciones de tamaño () pueden ser *O(n*). | 16 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 15 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 14 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 7 |
| Optimizador | *Minor* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 6 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 6 |
| Optimizador | *Minor* | No deben declararse múltiples variables en la misma línea  Declarar múltiples variables en una línea dificulta la lectura del código. | 5 |

**Proyecto**: EstadoSolicitudCredito

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 3 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 3 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 0 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 0 |
| Optimizadores | *Total* | 129 |
| *Blocker* | 0 |
| *Critical* | 20 |
| *Major* | 44 |
| *Minor* | 65 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 2 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 1 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 11 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 4 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener | 3 |
| Optimizador | *Critical* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 12 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 8 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.*   *StringBuilder* en lugar de *StringBuffer* | 6 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 5 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 3 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 3 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 2 |
| Optimizador | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 2 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 1 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 1 |
| Optimizador | *Major* | **Los bloques *Try-catch* no deben estar anidados**  Anidar *try/catch* bloques afecta gravemente a la legibilidad del código fuente porque hace demasiado difícil entender qué bloque atrapará cada excepción. | 1 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 28 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 10 |
| Optimizador | *Minor* |  | 8 |
| Optimizador | *Minor* |  | 4 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 4 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 3 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 2 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 1 |
| Optimizador | *Minor* | Las cláusulas "*catch*" deben hacer más actividades además de lanzar  Una cláusula de captura que sólo vuelve a lanzar la excepción capturada tiene el mismo efecto que omitir la captura por complete. Tales cláusulas deberían eliminarse o completarse con la lógica apropiada | 1 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 1 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 1 |
|  |  |  |  |

**Proyecto**: GarantiaInfonavit

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 0 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 0 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 0 |
| Optimizadores | *Total* | 103 |
| *Blocker* | 0 |
| *Critical* | 27 |
| *Major* | 30 |
| *Minor* | 46 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Optimizador* | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 18 |
| *Optimizador* | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 7 |
| *Optimizador* | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 2 |
| *Optimizador* | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 13 |
| *Optimizador* | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 6 |
| *Optimizador* | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 5 |
| *Optimizador* | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 2 |
| *Optimizador* | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 1 |
| *Optimizador* | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 1 |
| *Optimizador* | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 1 |
| *Optimizador* | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 1 |
| Optimizador | *Minor* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 17 |
| Optimizador | *Minor* | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 9 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 7 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 4 |
| Optimizador | *Minor* | **Las declaraciones "*switch*" deben tener por lo menos 3 cláusulas "*case*"**  Las declaraciones “*Switch*” son útiles cuando hay muchos casos diferentes dependiendo del valor de la misma expresión. Sin embargo, sólo en uno o dos casos, el código será más legible con sentencias *if*. | 2 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 2 |
| Optimizador | *Minor* | **Las declaraciones vacías deben ser eliminadas**  Las afirmaciones vacías, es decir las que se introducen generalmente por error deben eliminarse | 1 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 1 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 1 |
| Optimizador | *Minor* | **El paquete sin nombre (unnamed) por defecto no debe usarse**  Para hacer valer esta mejor práctica, las clases ubicadas en el paquete predeterminado ya no pueden accederse desde las clases nombradas desde *Java* 1.4. | 1 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 1 |
|  |  |  |  |

**Proyecto**: Precalificacion

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 6 |
| *Blocker* | 1 |
| *Critical* | 0 |
| *Major* | 5 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 5 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 5 |
| Optimizadores | *Total* | 711 |
| *Blocker* | 2 |
| *Critical* | 33 |
| *Major* | 80 |
| *Minor* | 596 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 1 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 4 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 1 |
| *Vulnerabilidad* | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 3 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador. | 1 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 1 |
| Optimizador | *Blocker* | **Los *Switch* *cases* deben terminar con una declaración incondicional "*break*"**  Cuando la ejecución no se termina explícitamente al final de un *switch* *case*, continúa ejecutando las sentencias del siguiente caso. Mientras que esto es a veces intencional, a menudo es un error que lleva a un comportamiento inesperado del aplicativo | 2 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 17 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 5 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 4 |
| Optimizador | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 4 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 1 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 23 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 17 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 11 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 6 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 4 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 4 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 3 |
| Optimizador | *Major* | **El uso de la función *String* debe optimizarse para caracteres individuales**  Una llamada a *indexOf* o *lastIndexOf* con una sola letra *String* se puede hacer más eficaz cambiando a una llamada con un argumento *char.* | 3 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 2 |
| Optimizador | *Major* | **Deben eliminarse los métodos "privados" no utilizados**  Métodos privados que nunca se ejecutan son código muerto, código innecesario e inoperante que debe ser eliminado. La eliminación del código muerto disminuye el tamaño de la base de códigos mantenida, lo que facilita la comprensión del programa y evita la introducción de errores. | 2 |
| Optimizador | *Major* | **Deben eliminarse los métodos "privados" no utilizados**  Métodos privados que nunca se ejecutan son código muerto, código innecesario e inoperante que debe ser eliminado. La eliminación del código muerto disminuye el tamaño de la base de códigos mantenida, lo que facilita la comprensión del programa y evita la introducción de errores. | 1 |
| Optimizador | *Major* | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. |
| Optimizador | *Major* | **Los bloques de multilínea deben estar encerrados en llaves**  Las llaves pueden omitirse de un bloque de una sola línea, por ejemplo con una sentencia *if* o para bucle, pero hacerlo puede ser engañoso e inducir a errores.  Esta regla plantea un problema cuando el espaciado en blanco de las líneas después de un bloque de una línea indica la intención de incluir esas líneas en el bloque, pero la omisión de las llaves significa que las líneas se ejecutarán incondicionalmente una vez. | 1 |
| Optimizaodor | *Minor* | **Los nombres de clase deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla permite comprobar que todos los nombres de clase coinciden con una expresión regular.  Con la expresión regular proporcionada por defecto *^[A-Z][a-zA-Z0-9]\*$:* | 244 |
| Optimizaodor | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 140 |
| Optimizaodor | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 127 |
| Optimizaodor | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 20 |
| Optimizaodor | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 14 |
| Optimizaodor | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 13 |
| Optimizaodor | *Minor* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 10 |
| Optimizaodor | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 8 |
| Optimizaodor | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 5 |
| Optimizaodor | *Minor* | Las declaraciones deben utilizar interfaces de *Java* *Collection* como "*List*" en lugar de clases de implementación específicas como "*LinkedList*"  El propósito de *Java* *Collections* *API* es proporcionar una jerarquía bien definida de interfaces para ocultar los detalles de implementación.  Las clases de implementación deben ser utilizadas para instanciar nuevas colecciones, pero el resultado de una instanciación debe ser almacenado idealmente en una variable cuyo tipo sea una interfaz de colección *Java*.  Esta regla plantea un problema cuando una clase de implementación:   * Es devuelto de un método público. * Se acepta como argumento para un método público. * Se expone como miembro público. | 4 |
| Optimizaodor | *Minor* | **Los nombres de clase deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla permite comprobar que todos los nombres de clase coinciden con una expresión regular.  Con la expresión regular proporcionada por defecto *^[A-Z][a-zA-Z0-9]\*$:* | 2 |
| Optimizaodor | *Minor* | **La devolución de expresiones booleanas no debe ser envuelta en una declaración *"if-then-else"***  Debe simplificarse la devolución de expresiones literales booleanas envueltas en enunciados *if-then-else*. | 2 |
| Optimizaodor | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 2 |
| Optimizaodor | *Minor* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 1 |
| Optimizaodor | *Minor* | **Las constantes públicas y los campos inicializados en la declaración deben ser "final estático (*static* *final*)" en lugar de meramente "*final*"**  Además, cuando un campo final no público no es también estático, implica que diferentes instancias pueden tener diferentes valores. Sin embargo, la inicialización de un campo final no estático en su declaración obliga a cada instancia a tener el mismo valor. Por lo tanto, estos campos deben ser estáticos o inicializados en el constructor. | 1 |

**Proyecto**: Quejas

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 13 |
| *Blocker* | 4 |
| *Critical* | 0 |
| *Major* | 9 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 3 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 3 |
| Optimizadores | *Total* | 233 |
| *Blocker* | 0 |
| *Critical* | 30 |
| *Major* | 75 |
| *Minor* | 128 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 4 |
| Bug | *Major* | **Los objetos no serializable no deben almacenarse en objetos" “*HttpSession*”**  Independientemente de si se decide serializar o no explícitamente la sesión, se puede escribir en disco de todos modos, ya que el servidor gestiona su uso de memoria en un proceso llamado "*passivation*".  Adicional, algunos servidores automáticamente escriben sus sesiones activas para archivar en el momento del cierre y anular la serialización de dichas sesiones al inicio. Aunque “*HttpSession*” no se extienda serializable, debe asumir que será serializado y en caso de no haber objetos no serializables almacenados en la sesión, se producirán errores. | 5 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 2 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 1 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 1 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 1 |
| *Optimizador* | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 21 |
| *Optimizador* | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 4 |
| *Optimizador* | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 3 |
| *Optimizador* | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 1 |
| *Optimizador* | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 1 |
| *Optimizador* | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 29 |
| *Optimizador* | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 14 |
| *Optimizador* | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 8 |
| *Optimizador* | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 6 |
| *Optimizador* | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 4 |
| *Optimizador* | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado | 4 |
| *Optimizador* | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 3 |
| *Optimizador* | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 2 |
| *Optimizador* | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 1 |
| *Optimizador* | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 1 |
| *Optimizador* | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 1 |
| *Optimizador* | *Major* | Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 1 |
|  |  |  |  |

**Proyecto**: Recompensas

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 1 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 7 |
| *Blocker* | 0 |
| *Critical* | 5 |
| *Major* | 0 |
| *Minor* | 2 |
| Optimizadores | *Total* | 78 |
| *Blocker* | 4 |
| *Critical* | 8 |
| *Major* | 46 |
| *Minor* | 20 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action* | 1 |
| Vulnerabilidad | *Critical* | **El código no debe ser inyectado y ejecutado dinámicamente**  La función de *eval* es una manera de ejecutar código arbitrario en tiempo de ejecución. La evaluación dinámica del código es lenta y un problema potencial de seguridad cuando los argumentos no han sido validados correctamente. | 5 |
| *Vulnerabilidad* | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 1 |
| *Optimizador* | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 4 |
| Optimizador | *Blocker* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 3 |
| Optimizador | *Blocker* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 1 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 21 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 9 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 5 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 3 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 3 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 2 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 1 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 1 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 1 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 9 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 5 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 2 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 2 |
| Optimizador | *Minor* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 1 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 1 |

**Proyecto**: RegistroMCI

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 15 |
| *Blocker* | 1 |
| *Critical* | 0 |
| *Major* | 11 |
| *Minor* | 3 |
|  | *Total* | 8 |
| *Blocker* | 3 |
| *Critical* | 0 |
| *Major* | 0 |
| *Minor* | 5 |
| Optimizadores | *Total* | 1563 |
| *Blocker* | 0 |
| *Critical* | 122 |
| *Major* | 550 |
| *Minor* | 891 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 1 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 4 |
| Bug | *Major* | **No se deben hacer verificaciones de igualdad**  Las comparaciones de diferentes tipos siempre dan como resultado *false*. Por lo que las comparaciones y el código dependiente debe eliminarse. Esto incluye:   * Comparando un objeto con *null.* * Comparando un objeto con un primitivo no emparentado (E. G. *string* con un int). * Comparación de clases no relacionadas. * Comparación de una clase e interfaz no relacionada. * Comparación de tipos de interfaz no relacionados. * Comparando un *array* con un *no-array*. * Comparando dos *arrays.*   Específicamente en el caso de los *arrays*, dado que los *arrays* no anulan *Objects.equals(),* llamar iguales en dos *arrays* es lo mismo que comparar sus direcciones. Esto significa que *array1*.*equals*(*array2)* es equivalente a *array1*===*array2*.  Sin embargo, algunos equipos de desarrollo pueden esperar que *Array.equals(Object obj*) haga más que una simple comparación de direcciones de memoria, comparando por ejemplo el tamaño y contenido de los dos *arrays*. En su lugar, == *operator* o *Arrays.equals(array1, array2*) siempre debe ser usado con *arrays*. | 3 |
| *Bug* | *Major* | **Las asignaciones no deben ser redundantes**  La propiedad transitiva indica que si A == B y B == C, entonces A == C. En tales casos, no es necesario asignar A a C o viceversa porque ya son equivalentes. Esta regla plantea un problema cuando una asignación es inútil porque la variable asignada ya contiene el valor en todos los trayectos de ejecución. | 2 |
| Bug | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 1 |
| Bug | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 1 |
| Bug | *Minor* | **Los operandos matemáticos deben unirse antes de la asignación**  Cuando una operación aritmética se realiza en números enteros, el resultado siempre será un número entero. Puede asignar ese resultado a un valor *long*, *double* o *float* con una conversión automática de tipo de dato, pero en caso de haber comenzado como *int* o *long*, es probable que el resultado no sea el esperado.  Por ejemplo, si el resultado de la división *int* se asigna a una variable de punto flotante, la precisión se habrá perdido antes de la asignación.  Para evitar resultados imprevistos por lo menos un operando debe ser lanzado o promovido al tipo de dato final antes de que la operación tenga lugar. | 3 |
| Vulnerabilidad | *Blocker* | **Las direcciones *IP* no deben estar en código duro**  Poner en duro dirección *IP* dentro del código fuente es una mala idea por varias razones:   * Se necesita compilar de nuevo si la dirección *IP* cambia. * Obliga a utilizar la misma dirección en todos los ambientes (Desarrollo, Pruebas y Producción). * Deja la responsabilidad de colocar el valor a utilizar en producción en manos del desarrollador. * Permite a los atacantes descompilar el código y descubrir direcciones potencialmente sensibles. | 5 |
| Vulnerabilidad | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 3 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 62 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 24 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener | 14 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 13 |
| Optimizador | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 5 |
| Optimizador | *Critical* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 3 |
| Optimizador | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 1 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 391 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 48 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 32 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 20 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 11 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 9 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 7 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 6 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 5 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 4 |
| Optimizador | *Major* | **Las variables locales no deberían ensombrecer los campos de clase**  Sombrear los campos con una variable local es una mala práctica que reduce la legibilidad del código. Hace confuso identificar si el campo o la variable están siendo utilizadas. | 3 |
| Optimizador | *Major* | **Un campo no debe duplicar el nombre de la clase que contiene**  Es confuso tener a un miembro de la clase con el mismo nombre (diferencias de caso aparte) que su clase cerrada. Esto es particularmente así cuando se considera la práctica común de nombrar una instancia de clase para la clase misma. La mejor práctica dicta que cualquier campo o miembro con el mismo nombre que la clase cerrada sea renombrada para ser más descriptivo del aspecto particular de la clase que representa o tiene. | 3 |
| Optimizador | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 3 |
| Optimizador | *Major* | **Las asignaciones no deben hacerse desde dentro de subexpresiones**  Las asignaciones dentro de las subexpresiones son difíciles de detectar y por lo tanto hacen que el código sea menos legible. Idealmente, las subexpresiones no deberían tener efectos secundarios. | 2 |
| Optimizador | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 2 |

**Proyecto**: SaldosyMov

A continuación, se muestran los resultados del análisis de calidad de código categorizado por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

| **Tipo** | **Severidad** | **Cantidad de Incidencias** |
| --- | --- | --- |
| *Bugs* | *Total* | 8 |
| *Blocker* | 0 |
| *Critical* | 0 |
| *Major* | 8 |
| *Minor* | 0 |
| Vulnerabilidades | *Total* | 10 |
| *Blocker* | 0 |
| *Critical* | 5 |
| *Major* | 0 |
| *Minor* | 5 |
| Optimizadores | *Total* | 371 |
| *Blocker* | 4 |
| *Critical* | 35 |
| *Major* | 66 |
| *Minor* | 266 |

A continuación se enlistan las incidencias del tipo Vulnerabilidad, Bugs y Optimizadores las cuales fueron identificadas por medio del análisis de calidad de código y que deben ser atendidas a la brevedad posible con el objetivo de reducir el riesgo de fallo durante la operación.

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Las asignaciones no deben ser redundantes**  La propiedad transitiva indica que si A == B y B == C, entonces A == C. En tales casos, no es necesario asignar A a C o viceversa porque ya son equivalentes. Esta regla plantea un problema cuando una asignación es inútil porque la variable asignada ya contiene el valor en todos los trayectos de ejecución. | 8 |
| Vulnerabilidad | *Critical* | **El código no debe ser inyectado y ejecutado dinámicamente**  La función de *eval* es una manera de ejecutar código arbitrario en tiempo de ejecución. La evaluación dinámica del código es lenta y un problema potencial de seguridad cuando los argumentos no han sido validados correctamente. | 5 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 3 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 1 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 1 |
| Optimizador | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 4 |
| Optimizador | *Critical* | **Los nombres de las propiedades no deben ser duplicados dentro de una clase u objeto**  *JavaScript* permite duplicar el nombre de las propiedades en las clases y objetos pero sólo la última instancia de un nombre duplicado determina el valor real que será utilizado. Por lo tanto, cambiar los valores de otras ocurrencias de un nombre duplicado no tendrá ningún efecto y puede causar errores.  Definir una clase con un constructor duplicado generará un error. | 23 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 4 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 3 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 3 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 2 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 15 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 13 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 12 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 10 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 3 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público | 3 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 3 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación | 2 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 1 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 1 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 1 |
| Optimizaodor | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 82 |
| Optimizador | *Minor* | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 63 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 40 |
| *Optimizador* | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 21 |
| *Optimizador* | *Minor* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 16 |
| *Optimizador* | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 12 |
| Optimizador | *Minor* | **Los nombres de clase deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla permite comprobar que todos los nombres de clase coinciden con una expresión regular.  Con la expresión regular proporcionada por defecto *^[A-Z][a-zA-Z0-9]\*$:* | 8 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 7 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 6 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 4 |
| Optimizador | *Minor* | ***Collection.isEmpty()* debe usarse para probar vacíos**  *Collection.size ()* se usa para probar los trabajos vacíos, pero el uso de *Collection.isEmpty* () hace que el código sea más legible y más eficaz. La complejidad temporal de cualquier implementación del método *isEmpty* () debe ser *O(1)* considerando que algunas implementaciones de tamaño () pueden ser *O(n*). | 2 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 2 |
| Optimizador | *Minor* | **Las declaraciones vacías deben ser eliminadas**  Las afirmaciones vacías, es decir las que se introducen generalmente por error deben eliminarse. | 1 |
| Optimizador | *Minor* | **"*toString()"* nunca debe llamarse en un objeto *String***  Invocar un método *designed* para devolver una representación de *string* de un objeto que ya es un *string* es poco útil. Esta construcción redundante puede ser optimizada por el compilador, pero mientras tanto será confusa. | 1 |
| Optimizador | *Minor* | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 1 |

por el tipo de incidencias encontradas y segmentado por el proyecto analizado:

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

El nivel de alineación entre la definición de la solución tecnológica generada en la etapa de Diseño de Soluciones y los artefactos generados en la etapa de Construcción es determinado con base en la siguiente tabla de niveles de conformidad:

| **Representación Gráfica de Alineación** | **Descripción de los Niveles de Alineación** | **Valor** |
| --- | --- | --- |
|  | La implementación no tiene características en común con el diseño de la solución | 1 |
|  | La implementación tiene algunas características en común con el diseño de la solución y tales características están implementadas en total conformidad con el diseño. Sin embargo, existen características en la especificación que no fueron implementadas y existen características implementadas que no son cubiertas en la especificación | 2 |
|  | Existe total concordancia entre el diseño de la solución y su implementación. Todas las características están implementadas conforme a la especificación y no existen características implementadas que no sean cubiertas por la especificación | 3 |

**Nota:** No se evaluó la alineación entre Diseño y Construcción de Soluciones Tecnológicas ya que no se generó documentación para la atención de la solicitud.

| **Aspecto Evaluado** | **Diseño** | **Construcción** | **Nivel de Alineación** |
| --- | --- | --- | --- |
| NA | NA | NA | NA |
| **Nivel de Alineación Final** | | | NA |

* 1. Consistencia entre Ambientes del Ciclo de Vida de Desarrollo

El nivel de consistencia entre Ambientes del Ciclo de Vida del Desarrollo es determinado con base en la siguiente tabla de niveles de conformidad:

| **Representación Gráfica de Alineación** | **Descripción de los Niveles de Consistencia** | **Valor** |
| --- | --- | --- |
|  | La tecnología evaluada no es consistente en términos de producto de *Software* utilizado, versión y *release* entre los 3 ambientes considerados en el ciclo de vida del desarrollo | 1 |
|  | La tecnología evaluada es consistente en términos de producto de *Software* utilizado, versión y *release* en 2 de los 3 ambientes considerados en el ciclo de vida del desarrollo | 2 |
|  | La tecnología evaluada es consistente en términos de producto de *Software* utilizado, versión y *release* en los 3 ambientes considerados en el ciclo de vida del desarrollo | 3 |

**Nota:** No se evaluó la alineación entre Diseño y Construcción de Soluciones Tecnológicas ya que no fue posible obtener información detallada sobre los ambientes utilizados.

| **Tecnología** | **Ambiente de Desarrollo** | **Ambiente de Pruebas** | **Ambiente de Producción** | **Nivel de alineación de ambientes** |
| --- | --- | --- | --- | --- |
| NA | NA | NA | NA | NA |
| **Nivel de Alineación Final** | | | | NA |

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

**Nota:** No se evaluó el cumplimiento Documental para Liberación en Ambiente Productivo ya que no se generó documentación para la atención de la solicitud.

| **Nombre del Documento** | **Entregado** | **No Entregado** |
| --- | --- | --- |
| Plan de trabajo detallado de la solución | NA | NA |
| Diseño detallado de la solución | NA | NA |
| Plan de Implementación (Instalación, pruebas, retorno) | NA | NA |
| Componentes tecnológicos de la solución | NA | NA |
| Acuerdo de Niveles de Servicio | NA | NA |
| Solicitud de formalización del acuerdo de control de atributos de acceso | NA | NA |
| Acuerdo de control de atributos de acceso | NA | NA |
| Solicitud de custodia de cuenta privilegiada | NA | NA |
| Formato de liberación de aplicación | NA | NA |
| Formato de excepción a políticas de seguridad | NA | NA |
| Carta de Aceptación de Pruebas | NA | NA |
| Pruebas de Ancho de Banda | NA | NA |
| Formato de Alta o Modificación de Procesos | NA | NA |
| Atención a Usuario Final, Procesos de Gestión y CASIA | NA | NA |
| Documentos Varios (Memoria Técnica) | NA | NA |
| 01 (Servicio) - Información General | NA | NA |
| 02 (Servicio) - Procedimiento de Operación | NA | NA |
| 03 (Servicio) - Procedimiento de Recuperación | NA | NA |
| 04 (Servicio) - Monitoreo Disponibilidad | NA | NA |
| 05 (Servicio) - Cédula de Respaldos | NA | NA |
| Formatos para Control en CMDB | NA | NA |
| Formato RG1000 | NA | NA |
| Catálogo de Servicios | NA | NA |
| Capacitación a las áreas de Soporte Técnico a la Producción, al usuario final, y a Operaciones | NA | NA |

* **Indicadores de Alineación Entre Diseño y Construcción de Soluciones**
  + **Nivel de alineación por aspecto evaluado**: El nivel de alineación determina el grado de apego entre el diseño de soluciones tecnológicas y su construcción. Se utiliza la tabla anterior para definir el nivel de alineación de los aspectos evaluados.
  + **Nivel de alineación final**: Se genera al obtener el promedio de los niveles de alineación obtenidos.

**Nota:** No se especifican Indicadores de Alineación entre Diseño y Construcción de Soluciones ya que no se generó documentación para la atención de la solicitud.

| Aspecto Evaluado | Nivel de Alineación |
| --- | --- |
| Aspecto 1 | NA |
| Aspecto 2 | NA |
| Aspecto 3 | NA |
| Nivel de alineación final | NA |

* **Indicadores de Consistencia entre Ambientes**
  + **Nivel de alineación por aspecto evaluado**: Se obtiene con base en los criterios definidos en la tabla anterior.
  + **Nivel de alineación final**: Se genera al obtener el promedio de los niveles de alineación obtenidos.

**Nota:** No se especifican Indicadores de Consistencia entre Ambientes ya que no fue posible obtener información detallada sobre los ambientes utilizados.

| Tecnología Evaluada | Nivel de Alineación |
| --- | --- |
| Tecnología 1 | NA |
| Tecnología 2 | NA |
| Tecnología 3 | NA |
| Nivel de alineación final | NA |

* **Cumplimiento Documental**
  + **Cumplimiento Documental para Liberación en Ambiente Productivo**: Identificar el nivel de cumplimiento de entrega documental para realizar la liberación de la solución tecnológica en ambiente productivo

**Nota:** No se especifican Indicadores de Cumplimiento Documental para Liberación en Ambiente Productivo ya que no se generó documentación para la atención de la solicitud.

| **Indicador** | **Valor** |
| --- | --- |
| Cumplimiento documental para liberación en ambiente productivo | NA |

* 1. Propuestas de Acciones de Mejora
* **Calidad de Código**

Se sugiere atender de forma inmediata, las incidencias de tipo **Vulnerabilidad** y las de tipo ***Bug*** y **Optimizadores** con severidad ***Blocker***, ***Critical*** con el objetivo de reducir el riesgo de fallo durante la operación. Una vez atendidas las incidencias con severidad más alta, se sugiere establecer un plan de mantenimiento con el objetivo de corregir las incidencias con menor nivel de severidad.

Se recomienda ampliamente el uso de la extensión del IDE, “**Sonar Lint**”. El *plugin* antes mencionado permite identificar incidencias desde la construcción de la solución.

* **Alineación entre Diseño y Construcción de Soluciones Tecnológicas**

No se generan propuestas de acción de mejor debido a que no fue posible evaluar la Alineación entre Diseño y Construcción de Soluciones Tecnológicas.

* **Consistencia entre Ambientes del Ciclo de Vida de Desarrollo**

No se generan propuestas de acción de mejor debido a que no fue posible evaluar la Consistencia entre Ambientes del Ciclo de Vida de Desarrollo.

* **Cumplimiento Documental para Liberación en Ambiente Productivo**

No se generan propuestas de acción de mejor debido a que no fue posible evaluar el Cumplimiento Documental para Liberación en Ambiente Productivo.

1. Conclusiones

Preservar la calidad en el ciclo de vida de las soluciones tecnológicas es un factor de gran relevancia para el Instituto ya que reduce el riesgo de una afectación durante la operación del negocio.

Con base en el resultado de la evaluación de soluciones tecnológicas plasmada en este documento se logran identificar brechas de calidad que deben ser atendidas con el objetivo de incrementar la calidad de las soluciones tecnológicas generadas dentro del Instituto. Las brechas especificadas en este documento sirven de referencia para evitar errores similares en futuros proyectos.