UNIVERSIDAD
NACIONAL
DE COLOMBIA

Análisis automático de código fuente aplicado en el software para comercio en línea: SAP Commerce Cloud

Roger Steven Ramírez Espejo

Universidad Nacional de Colombia

Facultad de ingeniería

Departamento de Ingeniería de Sistemas e Industrial

Bogotá, Colombia

20 de octubre de 2023

Análisis automático de código fuente aplicado en el software para comercio en línea: SAP Commerce Cloud

Roger Steven Ramírez Espejo

Trabajo final de maestría presentado como requisito parcial para optar al título de:

Magíster en Ingeniería - Ingeniería de Sistemas y Computación

Director:

Felipe Restrepo Calle, PhD.

Línea de Investigación:

Computación Aplicada

Grupo de Investigación:

Programming Languages and Systems - PLaS

Universidad Nacional de Colombia

Facultad de Ingeniería

Departamento de Ingeniería de Sistemas e Industrial

Bogotá, Colombia

20 de octubre de 2023

Dedicatoria.

”La idea del determinismo, que establece la necesidad de los actos del hombre y rechaza la absurda leyenda del libre albedrío, no anula en absoluto la inteligencia ni la conciencia del hombre, como tampoco la valoración de sus acciones.”

Vladimir Lenin

Agradecimientos

Quiero aprovechar este espacio para expresar mi más sincero agradecimiento a mi director de trabajo final, Felipe Restrepo Calle, por su invaluable orientación y apoyo a lo largo de todo el proceso de escritura y desarrollo de este proyecto. Su dedicación, paciencia y comprensión han sido fundamentales para el logro de este trabajo académico. Durante el tiempo que hemos trabajado juntos, el profesor Felipe ha demostrado un profundo conocimiento en el campo de estudio, brindándome una guía experta y ofreciéndome valiosas perspectivas que han enriquecido mi trabajo. Sus comentarios y sugerencias han sido fundamentales para mejorar la calidad de este trabajo final, así como para impulsar mi crecimiento académico y profesional.

Deseo expresar mi profundo agradecimiento a mi familia: mis padres, José y Cecilia y mis Hermanos Jenny, Carol y Cesar, quienes desde su juventud enfrentaron grandes desafíos para formar a nuestra familia en un país que tanto en la actualidad y en aquel entonces, presenta limitaciones significativas en cuanto a oportunidades de desarrollo y educación. Su perseverancia, dedicación y valentía allanaron el camino para que pudiera tener una vida más próspera y llena de oportunidades. Reconozco que mis logros alcanzados en el ámbito educativo no son solo míos, sino que son también el resultado del sacrificio y el apoyo incondicional de mis padres. La determinación y el esfuerzo incansable de mis padres me inspiraron a perseguir mis metas académicas y a superar cualquier obstáculo que se presentara en el camino.

También deseo expresar mi más sincero agradecimiento a mi hijo Matías y a mi mascota Peggy, por ser mi constante fuente de inspiración y apoyo incondicional. Su amor, alegría y presencia en mi vida ha sido fundamental para mantenerme motivado y superar los desafíos que he enfrentado. En los momentos más difíciles, su presencia ha sido un bálsamo que me ha dado fuerzas para seguir adelante.

Finalmente, quiero dar las gracias a todos los profesores, colegas, y alumnos que han participado en mis 18 años de formación académica y profesional. Sin su ayuda, no habría podido llegar hasta donde estoy hoy.

Resumen

Análisis automático de código fuente aplicado en el software para comercio en línea: SAP Commerce Cloud.

Este documento plantea un trabajo final de maestría para apoyar el proceso de migración de SAP Commerce a la nube pública de Microsoft Azure. Se destaca la importancia de la analítica de software y el análisis automático de código fuente para obtener datos concretos que respalden las decisiones de arquitectura, diseño y desarrollo. SAP Commerce Cloud, un producto utilizado para implementaciones de comercio en línea, presenta desafíos al ajustar y extender sus funcionalidades según las necesidades de cada cliente. Además, se menciona la falta de una herramienta automatizada que facilite la migración de SAP Commerce a la nube pública. En este contexto, se propone una estrategia que utiliza la herramienta de análisis de código fuente PMD para respaldar las migraciones a la arquitectura en la nube, y se evalúa su rendimiento a través de estudios de caso.

Palabras clave: analítica de software, análisis automático de código fuente, SAP Commerce, migración, nube pública, Microsoft Azure, PMD, estudios de caso.

Abstract

Automatic source code analysis applied in software for e-commerce: SAP Commerce Cloud.

This document contains a master's final work project to support the process of migrating SAP Commerce to the public cloud of Microsoft Azure. It highlights the importance of software analytics and automatic source code analysis in obtaining concrete data to support architecture, design, and development decisions. SAP Commerce Cloud, a product used for online commerce implementations, presents challenges in adjusting and extending its functionalities to meet each client's needs. Furthermore, the lack of an automated tool to facilitate the migration of SAP Commerce to the public cloud is mentioned. In this context, a strategy is proposed that utilizes the PMD source code analysis tool to support migrations to the cloud architecture, and its performance is evaluated through case studies.

Keywords: software analytics, automatic source code analysis, SAP Commerce, migration, public cloud, Microsoft Azure, PMD, case studies.

Este trabajo final de maestría fue calificado en septiembre de 2023 por el siguiente evaluador:

Jairo Hernán Aponte Melo Ph.D.

Profesor Facultad de Ingeniería

Universidad Nacional de Colombia

Índice general

Resumen vii

Abstract viii

Lista de Figuras xv

Lista de Tablas xvii

1. Introducción 1

1.1. Motivación . . . 2

1.2. Problema de investigación y pregunta de investigación . . . 3

1.2.1. Migración de infraestructura a implementaciones basadas en la nube . . . 3

1.2.2. Transformación de experiencia de usuario de Java a Spartacus Angular . . . 5

1.2.3. Oportunidad de profundización . . . 5

1.3. Objetivos . . . 5

1.3.1. Objetivo general . . . 5

1.3.2. Objetivos específicos . . . 5

1.4. Metodología . . . 6

1.4.1. Actividades desarrolladas . . . 9

1.4.2. Artefactos entregables. . . . 11

1.5. Estructura del trabajo final . . . 11

2. Análisis de código fuente y SAP commerce 13

2.1. Analítica de software . . . 13

2.2. Análisis automático de código fuente 14
2.2.1. Trabajos relacionados con análisis automático de código fuente 15
2.2.2. Herramientas para análisis automático de código fuente 17
2.3. Software empresarial para comercio en línea: SAP Commerce 18
2.3.1. Arquitectura SAP Commerce 19
2.3.2. Análisis automático de código fuente en SAP Commerce 21
2.4. Selección de módulos de SAP Commerce y herramientas para el análisis automático de código fuente 23
2.4.1. Criterios de selección de módulos de SAP Commerce 23
2.4.2. Selección de módulos de SAP Commerce 25
2.4.3. Análisis comparativo de herramientas para el análisis estático de código fuente en SAP Commerce 32

3. Estrategia de análisis automático de código fuente en SAP Commerce 35

3.1. Análisis de características incompatibles con SAP Commerce 35
3.1.1. Detección de uso de extensiones obsoletas 36
3.1.2. Incompatibilidades de SAP Commerce con arquitectura cloud 38
3.1.3. Detección de librerías de terceros 41
3.1.4. Detección de palabras clave reservadas de motores de base de datos SQL . . 43
3.2. Análisis de código Java 45
3.2.1. Buenas prácticas de código Java 45
3.2.2. Estilo código Java 49
3.2.3. Diseño Java 52
3.2.4. Documentación Java 55
3.2.5. Código propenso a errores Java 56
3.2.6. Multi-hilo Java 61
3.2.7. Rendimiento Java 62
3.2.8. Seguridad Java 64
3.3. Análisis de código Javascript 64
3.3.1. Buenas prácticas JavaScript 64

3.3.2. Estilo de código de JavaScript 65
3.3.3. Código propenso a errores de JavaScript 66

4. Implementación estrategia de análisis de código en SAP Commerce 67
4.1. Herramienta para el análisis de código fuente de SAP Commerce 67
4.1.1. Modulo genérico *app* 69
4.1.2. Modulo genérico *utilities* 71
4.1.3. Modulo genérico *pmd-commons* 74
4.1.4. Módulo genérico *buildSrc* 75
4.1.5. Módulo análisis de extensiones obsoletas *deprecated-extensions-analysis* . . . 75
4.1.6. Módulo análisis de incompatibilidades de SAP Commerce *commerce-incompatibilities-analysis* 77
4.1.7. Módulo análisis de librerías de terceros *third-party-libraries-analysis* 78
4.1.8. Módulo análisis de palabras clave de motores de base de datos *database-keyword-commerce-analysis* 79
4.1.9. Módulo análisis de código java *java-commerce-analysis* 80
4.1.10. Módulo análisis de palabras clave de motores de base de datos *javascript-commerce-analysis* 81
4.2. Flujo de ejecución de la herramienta 82
4.2.1. Instalación y ejecución 83
4.2.2. Inicialización 84
4.2.3. Análisis 88

5. Evaluación de la estrategia mediante estudios de caso de SAP Commerce 92
5.1. Elementos de los reportes de evaluación 93
5.2. Estudio de caso 1: SAP Commerce 123 94
5.3. Estudio de caso 2: Tienda en línea macromercado 101
5.4. Estudio de caso 3: Tienda en línea entre comercios de artículos electrónicos 110

6. Conclusiones y trabajo futuro 122

Referencias 125

A. Código ilustrativo de reglas de análisis de código fuente 130

A.1. Bloques de código reglas de Java . . . 131

A.1.1. Ejemplos de buenas prácticas de Java . . . 131

A.1.2. Ejemplos de estilo de código Java. . . . 151

A.1.3. Ejemplos de reglas de diseño Java. . . . 169

A.1.4. Ejemplos de reglas de documentación Java. . . . 185

A.1.5. Ejemplos de reglas propensas a errores Java. . . . 186

A.1.6. Ejemplos de reglas de multihilo Java. . . . 213

A.1.7. Ejemplos de reglas de rendimiento de Java. . . . 217

A.1.8. Ejemplos de reglas de seguridad de Java. . . . 226

A.2. Bloques de código reglas de JavaScript . . . 227

A.2.1. Ejemplos de buenas prácticas de JavaScript . . . 227

A.2.2. Ejemplos de estilo de código de JavaScript . . . 229

A.2.3. Ejemplos de código propenso a errores de JavaScript . . . 232

Índice de figuras

1-1. Fases metodológicas del trabajo final de Maestría 7

2-1. Relación analítica de software y análisis automático de código fuente. 15

2-2. Antecedentes análisis automático de código fuente. **Fuente:** Elaboración propia . . . 16

2-3. Diagrama de Arquitectura alto nivel SAP Commerce Cloud (SAP About, 2022). 19

2-4. Diagrama de Arquitectura de integraciones SAP Commerce Cloud (Ceron, 2022). . . . 21

2-5. Artículo *Measuring Code with Sonar* del portal oficial de **CX Works**. (MacWilliam, 2022). 22

2-6. Diagrama de decisión para selección de módulos de SAP Commerce. 24

2-7. Diagrama de módulos definidos dentro del alcance del trabajo final.
Azul: módulos candidatos a ser incluidos dentro del alcance del análisis.
Morado: Módulos mantenidos y administrados por SAP los cuales están fuera del alcance.
Amarillo: Productos independientes que se integran con SAP Commerce y por lo tanto también se encuentran fuera del alcance. 26

3-1. Diagrama de procesos de la estrategia de análisis automático de código fuente en SAP Commerce. 36

4-1. Diagrama de dependencias de Gradle del módulo app. 70

4-2. Diagrama de clases del módulo *app*. 71

4-3. Diagrama de dependencias de Gradle del módulo *utilities*. 71

4-4. Diagrama de clases del módulo *utilities*. 72

4-5. Diagrama de dependencias de Gradle del módulo *pmd-commons*. 74

4-6. Diagrama de clases del módulo *pmd-commons*. 74

4-7. Diagrama de dependencias de Gradle del módulo *deprecated-extensions=analysis*. . . . 76
4-8. Diagrama de clases del módulo *deprecated-extensions-analysis*. 76
4-9. Diagrama de dependencias de Gradle del módulo *commerce-incompatibilities-analysis*. . 77
4-10. Diagrama de clases del módulo *commerce-incompatibilties-analysis*. 78
4-11. Diagrama de dependencias de Gradle del módulo *third-party-libraries-analysis*. 78
4-12. Diagrama de clases del módulo *third-party-libraries-analysis*. 79
4-13. Diagrama de dependencias de Gradle del módulo *database-keyword-commerce-analysis*. 79
4-14. Diagrama de clases del módulo *database-keyword-commerce-analysis*. 80
4-15. Diagrama de dependencias de Gradle del módulo *java-commerce-analysis*. 81
4-16. Diagrama de clases del módulo *java-commerce-analysis*. 81
4-17. Diagrama de dependencias de Gradle del módulo *javascript-commerce-analysis*. 82
4-18. Diagrama de clases del módulo *javascript-commerce-analysis*. 82
4-19. Configuración de ejecución de referencia de la herramienta IntelliJ IDEA. 85
4-20. Diagrama de flujo de la fase de inicialización. 85
4-21. Diagrama de flujo de la fase de análisis en la ejecución. 88

Índice de tablas

1-1. Fases metodológicas . . . 7
1-2. Actividades a desarrollar . . . 9
2-1. Listado de herramientas para el análisis automático de código fuente . . . 18
2-2. Comparativo herramientas candidatas para análisis de código fuente en SAP Commerce 33
3-1. Reporte de versiones de librerías de terceros . . . 42
3-2. Funciones propias de los motores de base de datos soportados por SAP Commerce. . 44
3-3. Conjunto de reglas de buenas prácticas de Java . . . 45
3-4. Conjunto de reglas Estilo de codificación para Java . . . 50
3-5. Conjunto de reglas de diseño para Java . . . 52
3-6. Conjunto de reglas de documentación para Java . . . 55
3-7. Conjunto de reglas de detección de posibles errores para Java . . . 56
3-8. Conjunto de reglas de detección de multi-hilo para Java . . . 61
3-9. Conjunto de reglas de rendimiento para Java . . . 62
3-10. Conjunto de reglas de seguridad para Java . . . 64
3-11. Conjunto de reglas de buenas prácticas de JavaScript . . . 64
3-12. Conjunto de reglas de estilo de codificación para JavaScript . . . 65
3-13. Conjunto de reglas de código propenso a errores para JavaScript . . . 66
4-1. Herramientas usadas en la implementación de la herramienta de análisis de código . 68

1. Introducción

La analítica de software ayuda a obtener datos concretos de las implementaciones, que apoyen decisiones de arquitectura, diseño, y desarrollo (Buse and Zimmermann, 2012). La analítica de software entre otras cosas, usa el análisis automático de código fuente, el cual, mediante métricas permite determinar la coherencia, complejidad, mantenibilidad, y otras características (Binkley, 2007).

Por otra parte, SAP Commerce Cloud es un producto con más de 20 años de desarrollo, el cual tiene como objetivo, acelerar el proceso de implementación de comercio en línea, bajo distintos modelos de negocio: B2C (Negocio a cliente), B2B (Negocio a negocio), B2C - Telecomunicaciones, entre otros (Ceron, 2022). SAP Commerce contiene las siguientes funcionalidades por defecto: administración de contenido web, órdenes, inventarios, precios, descuentos, promociones, mercadeo, pagos, análisis de fraudes, etc. Estas funcionalidades varían de cliente a cliente, por lo cual, estas deben ajustarse y/o extenderse. Esto, implica un desafío para los Partners[1] y los clientes, a la hora de adoptar todas las funcionalidades que el producto ofrece (Ceron, 2022).

SAP ha hecho un esfuerzo de documentar las buenas prácticas de implementación (SAP Help, 2022), y motivar el uso del análisis automático de código fuente mediante la herramienta SonarQube (MacWilliam, 2022). No obstante, en los últimos años ha surgido un nuevo escenario, llevar al producto SAP Commerce a la nube pública de Microsoft Azure mediante el modelo de Plataforma como un servicio, y por ende, migrar a todos sus clientes a la nueva tecnología. Esto, implica un esfuerzo por parte de los consultores de SAP, Partners, y clientes; que mediante proyectos de actualización, y migración, realizan ajustes a sus implementaciones para adoptar la nueva infraestructura.

Teniendo en cuenta lo anterior, en este momento no existe una herramienta automatizada que permita apoyar el proceso de migración de SAP Commerce a la nube pública. Por tal motivo, este documento

[1] Partners es el nombre global que se le da a las casas de software que están autorizadas para implementar y personalizar los productos de SAP.

plantea una propuesta de trabajo final de maestría, que propone apoyar el proceso de migración de SAP Commerce a la nube pública de Microsoft Azure, mediante el uso del análisis automático de código fuente. Así, se busca facilitar este proceso de adopción de la nueva infraestructura.

1.1. Motivación

SAP Commerce es un producto que tiene casi 20 años de evolución, el cual posee una gran cantidad de módulos y funcionalidades, por ende, tiene una curva de aprendizaje muy larga. A pesar de usar tecnologías populares como Spring Framework y Java, posee particularidades tales como: lenguajes de dominio específico como ImpEx y Flexible Search, inyección de scripts mediante groovy, construcción de proyectos usando ant y maven, esto se suma a nuevas tecnologías añadidas como Spartacus, angular y la nube de Microsoft Azure. Esta heterogeneidad de tecnologías es difícil de comprender para equipos de desarrollo con poca experiencia.

En la sección 2.1 se menciona que los desarrolladores tienden a crear conceptos, basados en opiniones subjetivas y no en evidencias concretas de los mismos (Menzies and Zimmermann, 2013). Esta situación se vive también con el producto SAP Commerce.

SAP es consciente de esta problemática, por lo tanto, ha definido las buenas prácticas de implementación mediante las guías **Development guidelines** (SAP Help, 2022) y **CX Works** (MacWilliam, 2022). Estos insumos pueden ser consultados, con el fin evitar dificultades en las implementaciones del producto, re-uso de módulos antiguos o descontinuados, y el uso de malas prácticas de desarrollo de software.

SAP ha puesto en marcha un plan global, para llevar el producto SAP Commerce al modelo de Plataforma como un servicio en la nube de Microsoft Azure. Aunque existen herramientas para hacer análisis automático de código fuente sobre el producto, en el momento estas herramientas no soportan el proceso de migración a esta nueva arquitectura.

Teniendo en cuenta lo anterior, es oportuno que mediante este trabajo final de maestría se pueda crear una estrategia que permita apoyar este proceso de transición SAP Commerce, y facilitar su adopción por parte de los clientes.

1.2. Problema de investigación y pregunta de investigación

A pesar de que ya existe un esfuerzo en torno al análisis automático de código fuente en SAP Commerce (Sección 2.3.2), aún es necesario seguir trabajando en este campo. En especial, para apoyar la transición de este producto a la nube de Microsoft Azure, en el modelo de plataforma como un servicio. A continuación en las secciones 1.2.1 y 1.2.2 se profundiza en la situación actual del problema.

1.2.1. Migración de infraestructura a implementaciones basadas en la nube

En el año 2017, SAP tomó la decisión de implementar un modelo de negocio para SAP commerce llamado: “*Pay as you grow - Paga tanto como crezca tu negocio*”. Esta licencia permite que los clientes paguen por el número de órdenes (compras) que su sitio tiene, y la infraestructura es provista por SAP en la nube pública de Microsoft Azure mediante el modelo de Plataforma como un servicio: **Modelo Cloud**. Este licenciamiento se diferencia del modelo anterior, donde la licencia de SAP Commerce se vendía por separado de la infraestructura, por lo tanto, los clientes tenían la opción de usar los servidores de SAP o construir su propio centro de datos: **Modelo OnPrem**.
SAP determinó, que para finales del año 2028, todos los clientes con Licencia OnPrem deben ser migrados al modelo Cloud. Esto presenta un desafío para los Partners y clientes, toda vez, que se necesitan hacer los siguientes ajustes de código:

- **Ajustes personalizados para base de datos relacional:** Para el modelo OnPrem, los clientes podían elegir libremente la base de datos relacional de su preferencia (Oracle, SAP HANA, SQL Server, etc.). Sin embargo, el modelo Cloud sólo permite el uso de la base de datos Azure SQL Server. Aunque SAP Commerce dispone de “Flexible Search”[2], muchos Partners usan funciones propias del motor de base de datos. Por ejemplo, la función `DATETIME` de Oracle no existe en AzureSQL Server. Por lo tanto, se deben identificar y transformar esas funciones para evitar

[2] Lenguaje de dominio específico que permite ejecutar consultas SQL sobre los motores de base de datos soportados por SAP Commerce.

problemas una vez que la implementación haya sido migrada.

- **Configuraciones de infraestructura:** En la arquitectura OnPrem, los consultores y desarrolladores de SAP Commerce deben realizar configuraciones para la arquitectura, como el acceso a la base de datos, las configuraciones de clúster, las configuraciones de monitoreo, el registro y la caché (SAP Managed Properties, 2023). Sin embargo, muchas de estas configuraciones, que se encuentran en archivos de propiedades de Java, son manejadas automáticamente por la infraestructura de SAP en la nube. Por lo tanto, es necesario identificarlas y eliminarlas del código fuente.

- **Actualización de librerías de terceros:** SAP recomienda que las librerías de Java desarrolladas por terceros que se utilizan en las personalizaciones siempre se actualicen a la última versión disponible. Las actualizaciones de las bibliotecas contienen correcciones de errores, mejoras de rendimiento y arreglo de vulnerabilidades que pueden evitar problemas graves, como la brecha de seguridad de log4j que se informó en diciembre de 2021 (Blumenthal, 2022).

- **Revisión de buenas prácticas de Java y Javascript:** SAP Commerce utiliza varios lenguajes y herramientas propias, pero también se basa en los lenguajes de propósito de dominio Java y JavaScript. Los consultores de SAP también recomiendan que se identifiquen problemas o malas prácticas implementadas en estos lenguajes, y que se corrijan durante el proceso de migración a la nube.

Los puntos mencionados anteriormente son algunas de las revisiones y evaluaciones que realizan los consultores de SAP de forma manual con el fin de estimar el esfuerzo que requiere una migración de SAP Commerce a la nube. Algunas correcciones, como las palabras clave de base de datos y las incompatibilidades de SAP Commerce con la nube, son obligatorias debido a que el código no va a funcionar en la infraestructura en la nube. Las demás correcciones, como la actualización de librerías de terceros y la corrección de buenas prácticas de Java y JavaScript, son opcionales pero altamente recomendadas.

La realización de estas revisiones y evaluaciones permite a los consultores de SAP proporcionar a los clientes una estimación precisa del esfuerzo y el coste de la migración a la nube.

1.2.2. Transformación de experiencia de usuario de Java a Spartacus Angular

SAP Commerce para sus aceleradores usa la tecnología JSTL para el desarrollo de las plantillas visuales y la experiencia de usuario. No obstante, esta es una estrategia antigua fuertemente acoplada, que SAP decidió reemplazar por el framework basado en Angular **Spartacus**.
Spartacus tiene definido un conjunto de buenas prácticas, no obstante, no son analizadas en SonarQube como se observa en **Measuring code Quality with Sonar**. Por lo tanto, es necesario apoyar a los Partners y clientes, en la definición del conjunto de reglas para el análisis automático de código fuente en implementaciones que tengan Spartacus.

1.2.3. Oportunidad de profundización

Teniendo en cuenta los cambios generados por el objetivo de SAP para evolucionar el producto SAP Commerce Cloud a tecnologías basadas en la nube se presenta la siguiente oportunidad:

> *¿Cómo puede el análisis automático de código fuente apoyar la migración de Proyectos de SAP Commerce al modelo de Plataforma como un servicio que se encuentra en la nube de Microsoft Azure?*

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar una estrategia de análisis automático de código fuente para apoyar las implementaciones de SAP Commerce en su proceso de migración a la nube pública de Microsoft Azure.

1.3.2. Objetivos específicos

1. Seleccionar los módulos de SAP Commerce que aún no cuentan con revisión automática de código fuente y son parte de los proyectos de migración.

2. Diseñar una estrategia de análisis automático de código fuente para los módulos seleccionados anteriormente del producto SAP Commerce.

3. Implementar la estrategia diseñada mediante el uso de herramientas de análisis automático de código fuente.

4. Evaluar el funcionamiento de la estrategia propuesta mediante un estudio de caso de un proyecto de migración de SAP Commerce.

1.4. Metodología

El trabajo final de maestría se dividirá en 4 fases metodológicas, que tendrán distintos métodos de ejecución. Inicialmente comprenderá un análisis sobre los proyectos de migración del producto SAP Commerce a la nube pública, con el fin de definir cuáles serán los procesos que serán apoyados por el análisis automático de código fuente. Posteriormente, en la fase 2 se realizará un diseño de la estrategia para hacer análisis automático de código fuente sobre los módulos previamente seleccionados, mediante la caracterización y selección de herramientas disponibles en el mercado para ello, las cuales se escogerán mediante la ejecución de una prueba de concepto. En la fase 3 se usará una metodología de desarrollo en cascada para la implementación de la estrategia de análisis automático de código fuente. Por último, se hará la respectiva evaluación y documentación de esta estrategia mediante un estudio de caso, obtenido de una implementación de SAP Commerce (ver Figura **1-1**).

El tipo de estudio será mediante el uso de **diseño cuantitativo**, en donde bajo el marco metodológico del **Estudio de caso** se realizará la evaluación del proyecto. El estudio de caso "*implica el detallado e intensivo análisis de un caso único, en donde se analiza la complejidad particular del caso y su naturaleza*" (Bryman, 2016).
En este proyecto se aplicarán los lineamientos del **tipo de investigación aplicada** en donde se toman como insumos de entrada: Herramientas existentes para el análisis automático de código fuente, evaluaciones previas de proyectos de migración por parte de los consultores de SAP Commerce, y el trabajo previo del análisis automático de código fuente sobre SAP Commerce, con el fin de diseñar una estrategia que aporte a los proyectos de migración.

Figura **1-1**.: Fases metodológicas del trabajo final de Maestría

Teniendo en cuenta este tipo de estudio, a continuación en la Tabla **1-1** se describen con mayor detalle las fases metodológicas.

Tabla **1-1**.: Fases metodológicas

Fases metodológicas
Fase 1 - Identificación y análisis: Identificar módulos de SAP Commerce para análisis automático de código fuente.
Descripción: El producto SAP Commerce posee muchas funcionalidades, integraciones, y casos de uso de caja (ver Figura **2-4**), por lo tanto, es necesario acotar el alcance para el cual se desea aplicar análisis de código fuente.

Teniendo en cuenta lo anterior, esta fase se concentrará en definir cuáles son los módulos y funcionalidades, que son parte de los proyectos de migración del producto SAP Commerce a la nube pública de Microsoft Azure, y puedan beneficiarse del análisis automático de código fuente. Esta fase define los criterios de selección de los módulos de SAP Commerce en el contexto de los proyectos de migración. Posteriormente, a través de los criterios previamente definidos se seleccionarán los módulos de SAP Commerce para definir el dominio de trabajo sobre el producto. Los criterios de selección se definirán mediante una revisión documental de los servicios de "*Evaluación de proyectos de migración*", que ejecutan los consultores de SAP Commerce al inicio de un nuevo proyecto de migración.
Fase 2 - Diseño de estrategia: Diseñar una estrategia de análisis automático de código fuente sobre SAP Commerce.
Descripción: Esta fase se encargará del diseño de la estrategia que se implementará y evaluará en las fases posteriores. Inicialmente se caracterizarán y compararán las herramientas existentes para el análisis automático de código fuente. Posteriormente, se seleccionará una de ellas para ser aplicada en el producto SAP Commerce. Con el fin de garantizar que la herramienta seleccionada es la correcta, se hará una prueba de concepto sobre un módulo de SAP Commerce. Adicionalmente, esta prueba de concepto ayudará a tomar decisiones de diseño sobre la estrategia de análisis automático de código fuente y su implementación.
Fase 3 - Implementación: Implementar la estrategia diseñada.
Descripción: Esta fase de implementación será ejecutada con base en el diseño producto de la fase anterior. Inicialmente se definen los casos de uso que cumplirá la implementación, los cuales generarán las tareas que serán parte de la planeación. Posteriormente, se diseñarán los componentes técnicos de la estrategia, que serán implementados durante el desarrollo y posteriores pruebas de la estrategia.
Fase 4 - Evaluación: Evaluar la estrategia implementada sobre SAP Commerce.
Descripción: Una vez la fase de implementación concluya, se evaluará la estrategia sobre un estudio de caso. En primera instancia se definirán las métricas de evaluación de la estrategia, posteriormente se planteará un estudio de caso sobre el que se implementará la estrategia. El autor (Bryman, 2016) plantea en su libro *Social Research methods*, que existen distintos tipos de caso: crítico, extremo o único, representativo o típico, revelador, y longitudinal. El **Caso representativo o único**, también llamado **caso ejemplar**, tiene como objetivo "*capturar las circunstancias y condiciones de una situación cotidiana o común.*" Por lo tanto, los proyectos de migración son un caso representativo, toda vez que se están ejecutando para la gran mayoría de los clientes de SAP Commerce alrededor del mundo. Así, representar un caso particular, genera una estrategia que puede ser aplicada en los demás proyectos de migración. Teniendo en cuenta lo anterior se probará la estrategia sobre un estudio de caso, documentando sus resultados y concluyendo el trabajo, planteando los hallazgos y definiendo el trabajo futuro, en el reporte de memoria de trabajo final de maestría.

Fuente: Elaboración propia

1.4.1. Actividades desarrolladas

En esta sección se listan todas las actividades a desarrollar, teniendo en cuenta que cada una de las actividades corresponde a una fase de la metodología, las fases se nombran así: Identificación y análisis, Diseño de estrategia, Implementación, Evaluación. Cada actividad corresponde a esta denominación mediante listado numérico en orden de ejecución con respecto a los objetivos. En la Tabla **1-2** se evidencian las actividades de cada fase.

Tabla **1-2**.: Actividades a desarrollar

Fase	Objetivo Específico	Actividad	Descripción
Identificación y Análisis	1	1.1 Definición de criterios de selección de los módulos de SAP Commerce.	En esta actividad se establecerán los criterios de evaluación de los módulos de SAP Commerce, los cuales son parte del proceso de migración y se benefician del análisis automático de código fuente.
		1.1 Selección de módulos con base en los criterios de selección.	Seleccionar los módulos de SAP Commerce a los cuales se les aplicará análisis automático de código fuente, mediante los criterios de evaluación presentados en la sección anterior
Diseño de estrategia	2	2.1 Caracterización de herramientas y métricas de análisis automático de código fuente.	Caracterizar las herramientas y métricas para el análisis automático de código fuente existentes en la literatura, determinar su uso y compararlas con el fin de identificar los posibles insumos que serán parte del diseño de la estrategia.
		2.2 Selección de herramienta para el análisis automático de código fuente sobre SAP Commerce.	Seleccionar la herramienta para el análisis automático de código fuente que sea relevante para los módulos de SAP Commerce definidos en la fase anterior y que pueda apoyar el análisis automático de código fuente.

Fase	Objetivo Específico	Actividad	Descripción
		2.3 Evaluación de herramienta seleccionada mediante una prueba de concepto.	Una vez seleccionada la herramienta en la actividad anterior, se confirma la viabilidad de la herramienta mediante una prueba de concepto sobre uno de los módulos de SAP Commerce.
		2.4 Diseño de estrategia de análisis automático de código fuente.	Diseñar la estrategia de análisis automático de código fuente con base en los hallazgos encontrados por la prueba de concepto para aplicarlos en todos los módulos de SAP Commerce seleccionados.
Implementación	3	3.1 Definición de requerimientos y caso de uso para la estrategia.	Definir y documentar los requerimientos que tendrá la estrategia, orientados al análisis automático de código fuente sobre los módulos de SAP Commerce.
		3.2 Planeación de la implementación.	Esta actividad define el tiempo de ejecución de las actividades posteriores dentro de esta fase para la implementación de la estrategia.
		3.3 Diseño de la implementación.	Diseñar los componentes técnicos de la estrategia mediante diagramas de flujo, UML o de procesos.
		3.4 Desarrollo e implementación de la estrategia.	Implementar los componentes técnicos de la estrategia teniendo en cuenta el plan y el diseño producto de las fases anteriores.
		3.5 Prueba de implementación	Probar la estrategia desde el punto de vista técnico, con el fin de identificar y corregir errores en la implementación.
Evaluación	4	4.1 Definición de métricas de evaluación.	Se definen cuáles serán los criterios de evaluación de la estrategia.
		4.2 Planteamiento de estudio de caso.	Plantear y describir el estudio de caso de una implementación de SAP Commerce, sobre el cual se aplicará la estrategia de análisis automático de código fuente.
		4.3 Prueba de estrategia sobre estudio de caso.	Probar la estrategia sobre el estudio de caso planteado anteriormente con el fin de evaluar la métricas definidas en la primera actividad de esta fase.
		4.4 Presentación de resultados.	Presentar resultados de la evaluación de la estrategia sobre el estudio de caso para evaluar la validez de la misma.

Fase	Objetivo Específico	Actividad	Descripción
		4.5 Reporte memoria de trabajo final de maestría.	Reportar y recopilar todos los artefactos del trabajo final de maestría, que serán presentados como resultado final de este proyecto. También, se presentarán las conclusiones del trabajo y se plantearán las opciones de trabajos futuros complementarios.

Fuente: Elaboración propia

1.4.2. Artefactos entregables.

Los entregables de la Tabla **1-3** enumeran los artefactos que se generarán producto de la ejecución del trabajo final y las actividades mencionadas en la Sección 1.4.1

Tabla **1-3**.: Enumeración de entregables del desarrollo del trabajo final.

Lista de entregables	
#	Descripción de entregable
1	Documento con los módulos del producto SAP Commerce que serán usados para aplicar análisis de código fuente.
2	Documento con el resultado del análisis comparativo de herramientas de análisis automático de código fuente.
3	Reporte prueba de concepto de análisis de código fuente sobre SAP Commerce.
4	Documento con la definición de requerimientos del análisis automático de código fuente sobre SAP Commerce.
5	Documento con el diseño de la estrategia de análisis de código fuente para SAP Commerce.
6	Código fuente y documentación de la estrategia de análisis de código fuente.
7	Documento con planteamiento y resultados del estudio de caso.
8	Memoria de trabajo final de maestría con la documentación del trabajo realizado.

Fuente: Elaboración propia

1.5. Estructura del trabajo final

El resto del documento está estructurado de la siguiente forma:

- **Capítulo 2**: Se presenta el marco conceptual del análisis automático de código fuente y el software SAP Commerce. Se hace una descripción del mismo, junto con la selección de módulos de la solución que se van a analizar junto con el trabajo que se ha hecho de análisis de código para esta solución. Por último en este capítulo se hace una evaluación de las herramientas de análisis automáticos de código fuente que se pueden aplicar para las implementaciones de SAP Commerce. Este capítulo desarrolla en detalle el objetivo específico número 1.
- **Capítulo 3**: Se narra la estrategia de análisis automático de código fuente que se usará para evaluar las implementaciones de SAP commerce en el contexto de proyectos de migración. En este capítulo se presenta la estrategia desde el punto de vista conceptual sin entrar en detalles en la implementación. Este capítulo desarrolla en detalle el objetivo específico número 2.
- **Capítulo 4**: Describe la implementación de la estrategia presentada en el capítulo anterior, donde se presentan las tecnologías usadas, los detalles a alto nivel de la implementación y el flujo de ejecución y análisis de la herramienta. Este capítulo desarrolla en detalle el objetivo específico número 3.
- **Capítulo 5**: Se hace un planteamiento de 3 distintos estudios de caso con los cuales se evalúa la implementación, donde se presentan los resultados encontrados por la herramienta y su respectivo análisis. Este capítulo desarrolla en detalle el objetivo específico número 4.
- **Capítulo 6**: Contiene las conclusiones del trabajo final con la propuesta de trabajo futuro.

2. Análisis de código fuente y SAP commerce

En este capítulo se describe el estado del arte del proyecto, donde se definen y se exploran los antecedentes de la analítica de software (Sección 2.1). Posteriormente, se describe el análisis automático de código fuente (Sección 2.2). Adicionalmente, se explora a alto nivel las características del software empresarial SAP Commerce (Sección 2.3), el cual es el producto focal de este trabajo final de maestría y se presentan los antecedentes a nivel de análisis automático de código fuente en este producto. Por último, se realiza la selección de los módulos de SAP Commerce que serán objeto de estudio en este trabajo y se seleccionan las herramientas para realizar el proceso de análisis automático de código de los módulos seleccionados (Sección 2.4).

2.1. Analítica de software

La analítica de software se puede entender como el proceso de análisis de datos de implementaciones de software para gerentes e ingenieros, que tiene como objetivo capacitar a los equipos de desarrollo para obtener y compartir información concreta de la construcción del software, para tomar mejores decisiones de arquitectura, diseño e implementación (Buse and Zimmermann, 2012). Esta es una excelente herramienta para descubrir, verificar y monitorear los factores que afectan el desarrollo de software (Menzies and Zimmermann, 2018).

La analítica de software descubrió que los ingenieros y desarrolladores desarrollan sus ”*propias ideas*” de buen o mal software en la mayoría de los casos basados en la percepción y la subjetividad obtenida de proyectos anteriores, y no en una evidencia concreta de los mismos (métricas). Esto puede

llevar a que ideas anticuadas y malas prácticas sean reusadas en nuevas implementaciones (Menzies, 2019). La premisa anterior se comprueba en el artículo *Belief & Evidence in empirical software engineering* (Devanbu et al., 2016), cuyo objetivo fue el de hacer un experimento con 564 desarrolladores de diferentes niveles de experiencia, en donde se les pidió que hicieran un análisis de la calidad de una implementación de software. Posteriormente mediante una encuesta, se les pidió evaluar la calidad de código, la documentación, la probabilidad de tener defectos en el futuro, entre otras características. Estas respuestas se compararon con los resultados obtenidos mediante analítica de software, donde se nota una gran diferencia entre lo evaluado subjetivamente por los desarrolladores versus los datos obtenidos mediante la analítica de software. A través de este estudio se demostró que:

1. Los programadores tienen creencias muy fuertes en ciertos temas, inclusive algunos tienen favoritismos con respecto a algunas tecnologías específicas.

2. Estos favoritismos y creencias se forman con base en experiencias pasadas y/u opiniones de pares, en vez de un análisis riguroso de datos concretos.

3. Los conceptos subjetivos de los desarrolladores varían de proyecto a proyecto, pero no representan ninguna evidencia concreta de los mismos.

2.2. Análisis automático de código fuente

El análisis automático de código fuente es un sub-campo perteneciente a la analítica de software (ver Figura **2-1**), el cual provee datos y estadísticas que permiten determinar aspectos tales como: Coherencia lógica de la implementación, uso de buenas prácticas, y complejidad ciclomática (Binkley, 2007). Este campo parte de la premisa que cualquier aplicación implementada en cualquier lenguaje de programación, necesita adherirse sintácticamente a unos estándares validados directamente por el compilador. No obstante, aunque los programas son sintácticamente correctos, en muchos casos son lógicamente incorrectos o implementan malas prácticas que pueden generar problemas de rendimiento, seguridad, mantenibilidad y coherencia; aspectos que generalmente son ignorados por el compilador (Lathar et al., 2017).

Figura **2-1**.: Relación analítica de software y análisis automático de código fuente.

Este campo es usado para múltiples aplicaciones tales como: recuperación de diseño y arquitectura, detección de clones, detección de fallas, desarrollo basado en modelos, optimización de técnicas de ingeniería de software, análisis de rendimiento, evolución de programas, entre otros (Binkley, 2007).

2.2.1. Trabajos relacionados con análisis automático de código fuente

En la Figura **2-2**, se puede observar la evolución del análisis automático de código fuente, desde sus inicios hasta la actualidad. La primera fase (1950-1990) corresponde a los **primeros antecedentes** donde se identifica el problema (Wilkes, 1985), se relaciona la cantidad de código fuente con la probabilidad de que tenga defectos (Akiyama, 1971). Posteriormente, se define el concepto de **complejidad ciclomática** (McCabe, 1976), el cual corresponde al número de caminos independientes que puede tener una aplicación, generados por condicionales `if`, `switch` o ciclos `for`, `while`, `do`. Luego con la aparición de gestores de código fuente como VAX* DEC*/CMS (Anderson, 1987) se plantea la necesidad de mantener el control de cambios para identificar la aparición de posibles errores.

Durante las décadas del 90 y 2000 empezamos a ver la etapa **fundación de métricas y el auge de herramientas** que serían usadas extensiblemente durante la siguiente fase. Por ejemplo, en (Ohlsson et al., 1998) se hace una definición y expansión de métricas en la compañía Ericsson. Estas métricas también se posicionaron y fortalecieron con el lanzamiento de herramientas como FindBugs de la Universidad de Maryland (Ayewah et al., 2008) en Septiembre de 2006 y SonarQube de la compañía Sonar Source en el año 2007 (SonarSource, 2022). Estas herramientas tienen el objetivo de proveer

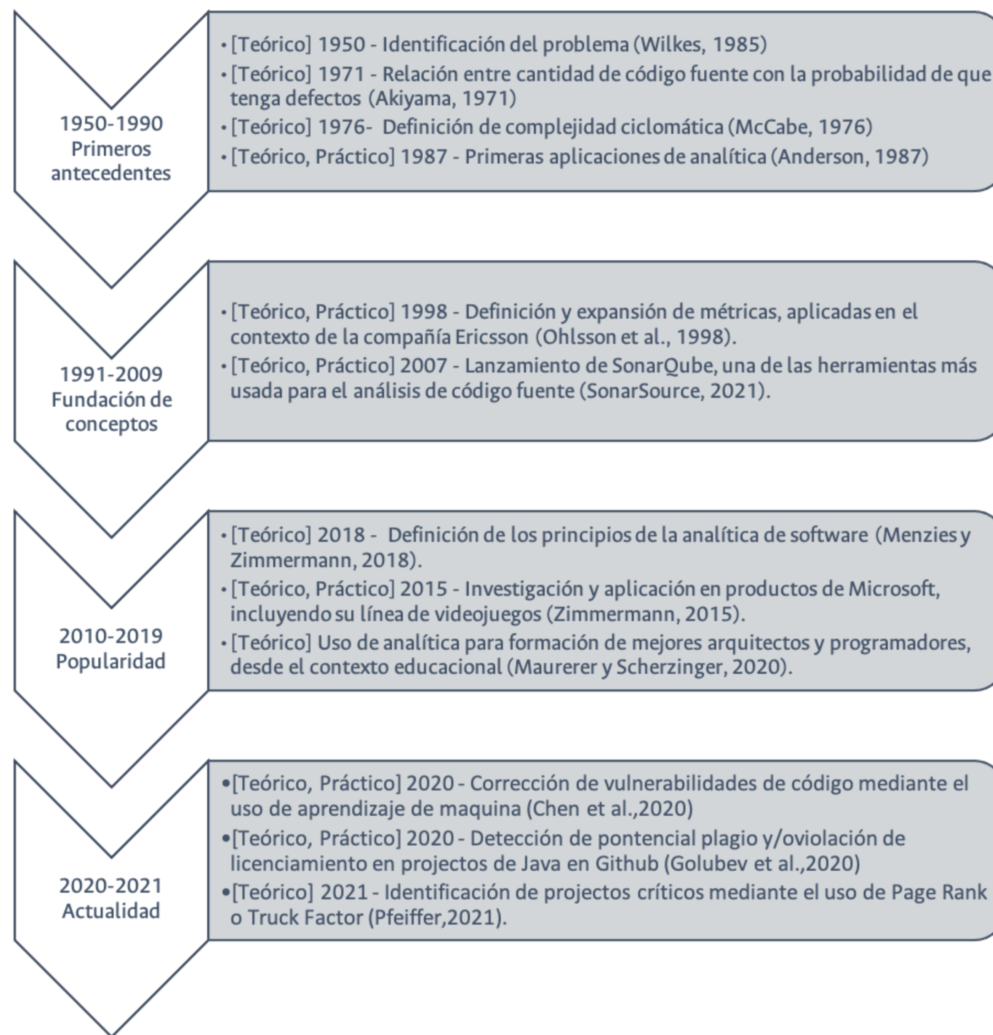

Figura **2-2**.: Antecedentes análisis automático de código fuente. **Fuente:** Elaboración propia

.

a los desarrolladores la capacidad de medir la calidad de código de sus proyectos a tal punto que: “*Realizar la inspección de código, debe ser tan popular como la integración continua*” (SonarSource, 2022). Una vez que el análisis automático de código fuente se establece como una necesidad para garantizar la calidad, correctitud, y mantenibilidad de las implementaciones de software, durante la década de 2010s, aparecen referentes como (Menzies and Zimmermann, 2013) que establecen los principios de la analítica de software y el análisis automático de código fuente. Esta etapa de **popularidad** genera proyectos e investigaciones en distintas áreas, por ejemplo: **videojuegos**, Thomas Zimmerman aplica estos principios en el desarrollo de la línea de video juegos por parte de Microsoft (Zimmer-

mann, 2015); **desarrollo de software sostenible**, (Anwar and Pfahl, 2017) realizan mediciones para determinar que los proyectos de software con defectos, vulnerabilidades, y bajo rendimiento son más propensos a usar más energía; **academia,** (Robles and Gonzalez-Barahona, 2013) utilizan PyChecker y PyLint para detectar posibles plagios y evaluar la calidad de código en las entregas de los estudiantes de ingeniería, adicionalmente, (Mauerer and Scherzinger, 2020) plantean un modelo empírico para la detección de errores comunes para educar mejores desarrolladores y/o arquitectos.

En la **actualidad**, el análisis automático de código fuente sigue siendo de gran interés, como se evidencia en las conferencias: **Minado de repositorios de Software 2021**, y **Manipulación y análisis de código fuente 2021**, donde se presentaron aportes al mejoramiento de la construcción de software. Por ejemplo: Corrección de vulnerabilidades de código mediante el uso de aprendizaje de máquina (Chen et al., 2020), Detección de potencial plagio y/o violación de licenciamiento en proyectos de Java en Github (Golubev et al., 2020) o la identificación de proyectos críticos mediante el uso de PageRank o Truck Factor (Pfeiffer, 2021).

2.2.2. Herramientas para análisis automático de código fuente

La analítica de código fuente ha sido objeto de observación desde la década de 1970, con una popularidad ascendente desde la segunda mitad de la década de 2000s hasta la actualidad. Por tal motivo existe un conjunto de herramientas para realizar análisis automático de código fuente. En la Tabla **2-1** se listan algunas de las más importantes.

Las aplicaciones resaltadas con **negrilla** en la tabla han sido usadas en trabajos recientes de analítica de código fuente. Por ejemplo **Sonarqube / SonarLint** es la herramienta más usada actualmente en especial para evaluar la calidad del código fuente, de tal forma que algunos contratos de implementación contienen cláusulas, que miden la calidad de los entregables basados en métricas obtenidos por esta herramienta (Antal et al., 2018; Huijgens et al., 2018; Pruijt et al., 2013; Raibulet and Arcelli Fontana, 2018).

Por otra parte **Frama-C** ha sido usada como objeto de estudio para determinar la calidad de software escrita en los lenguajes de C y C++ (Blanchard et al., 2019; Cuoq et al., 2012; Signoles, 2015). Así mismo, en (Soltanifar et al., 2016) se identifica el patrón de diseño implementado en Java, mediante

Tabla **2-1**.: Listado de herramientas para el análisis automático de código fuente

Herramientas	**Lenguajes soportados**
ConQAT AA	Java, C#, C++, JavaScript, ABAP, Ada y otros lenguajes.
dTangler	Java, Ruby
Sonarlint /Sonarqube	Java, Kotlin, C#, Visual Basic, C, C++, Javascript, Typescript, PHP, Python, Terraform, cloudformation, ABAP, Apex, Cobol, CSS3, Flex, Go, HTML5, Objective C, PL/I, PL/SQL, RPG, Ruby, Scala, Swift, TSQL, VB6, XML. (SonarSource, 2022)
Structure 101	C/C++, Delphi Pascal, Objective-C, Python, Java, .NET (C#)
Frama-C	C, C++, C#
Q-Rapids	Python y R
Smartshark	Python
RDFizer	Python, Java y R
Findbugs	Java

Fuente: Elaboración propia.

el uso de la predicción de defectos de los malos olores (*code smells*).

2.3. Software empresarial para comercio en línea: SAP Commerce

La analítica de software, como se especifica en la Sección 2.1, provee elementos para que los participantes de proyectos de software tengan herramientas tangibles para tomar mejores decisiones de diseño, arquitectura, e implementación de los proyectos.

A lo largo de esta sección se explicarán las características de alto nivel de la arquitectura (sección 2.3.1) que define la solución de SAP Commerce cloud. El objetivo de esta sección es dar contexto sobre la solución y no tiene la intención de profundizar en cada uno de los elementos de ella. Por otra parte, en la Sección 2.3.2 se presentan los antecedentes con respecto a análisis automático de código fuente en SAP Commerce.

2.3.1. Arquitectura SAP Commerce

SAP Commerce es una herramienta altamente flexible, la cual contiene un conjunto de librerías y herramientas extensibles implementadas en el Framework Spring (Spring, 2020), para la administración y gestión de una tienda de comercio en línea. La flexibilidad de SAP Commerce está definida por una arquitectura que contiene distintas capas abstractas y un conjunto de características modulares (SAP About, 2022) (ver Figura **2-3**).

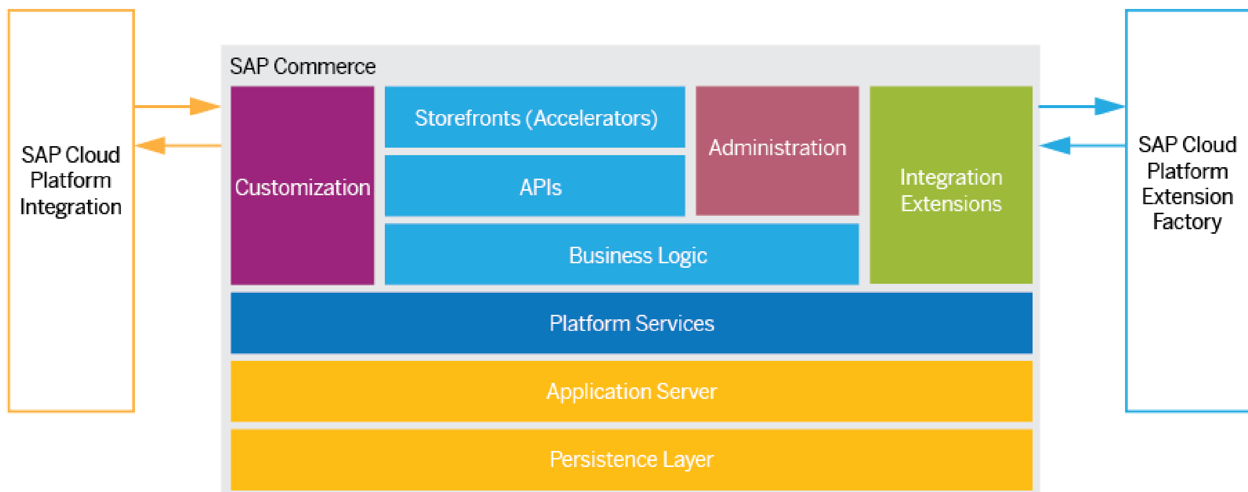

Figura **2-3**.: Diagrama de Arquitectura alto nivel SAP Commerce Cloud (SAP About, 2022).

- **Módulo de plataforma (Platform)**: El módulo de plataforma es el corazón de SAP Commerce, tal y como el Kernel lo es para los sistemas operativos Linux. Es el núcleo de la aplicación el cual contiene las funcionalidades usadas por todos los demás módulos y extensiones. Incluye soporte para la persistencia, registro, almacenamiento en caché, trabajos programados, seguridad, búsqueda, entre otros. Además, hay muchas extensiones opcionales que se pueden incluir para ampliar su funcionalidad y brindar un mejor soporte para los módulos comerciales proporcionados o para las nuevas funciones personalizadas que se deseen desarrollar (SAP, 2020).

- **Módulos y extensiones**: Los módulos y extensiones contienen lógica de negocio, APIs, y capas de presentación que agregan una funcionalidad específica. Un módulo consiste en una o

más extensiones que proveen diferentes funcionalidades para un caso de uso particular. Por ejemplo, un módulo puede agregar las siguientes funcionalidades:

- Proveer modelos de datos y lógica de negocio personalizada.
- Personalización de objetos de transferencia de datos (DTOs).
- Extensión de módulos administrativos de Backoffice.

Las extensiones pueden proveer servicios de tipo REST o AddOns (Adicionales). Los AddOns son extensiones "*instalables*" y "*removibles*" las cuales extienden la funcionalidad del Storefront[1], sin modificar el directamente el código ya existente. Adicionalmente, los módulos pueden usar y/o agrupar la lógica de negocio implementada en otros módulos. Estos módulos se construyen y compilan de forma transitiva cuando se detecta una relación de dependencia entre ellos.

- **Aceleradores**: SAP Commerce Cloud proporciona los "*aceleradores*", los cuales contienen características predefinidas: contenido web, plantillas, procesos de negocio, estructura de datos, entre otros, para diferentes industrias tales como: Venta de negocio a cliente (B2C), negocio a negocio (B2B), venta de servicios de telecomunicaciones (Telco), Banca e instituciones financieras (Financial), Viajes (Travel), Acelerador de servicios gubernamentales (Citizen engagement) y Venta de negocio al cliente para el mercado Chino[2]. Como su nombre lo indica los aceleradores, tienen el propósito de apresurar el proceso de implementación de una tienda en línea, con el fin de reducir el tiempo y costo de la implementación, por ende, agilizar el proceso de salida a producción.
- **Integración y personalización**: Cualquier aspecto o módulo de SAP Commerce puede ser extendido o modificado en aras de proveer soporte para cualquier escenario requerido. Adicionalmente, SAP Commerce provee integraciones para replicar datos hacia aplicaciones de terceros.
- **Configuración y administración**: SAP Commerce provee herramientas para la administración de datos, flujos de negocio, usuarios, roles, permisos, órdenes, precios, inventario, etc., y cual-

[1] El término "*Storefront*" es el término común dentro de SAP Commerce para referirse a la capa de presentación (interfaz gráfica) de la tienda en línea, usada por los clientes para adquirir productos.

[2] Por regulaciones gubernamentales especiales del mercado chino, es necesario considerar implementaciones de un acelerador que soportara el mercado Chino.

quier otro dato administrado por la aplicación mediante la herramienta de Backoffice[3] . Estas herramientas de administración, como todas las extensiones y módulos de Commerce, son personalizables y extensibles.

2.3.2. Análisis automático de código fuente en SAP Commerce

Los clientes no sólo usan SAP Commerce para vender sus productos, sino que además, desean optimizar sus procesos operacionales, mediante integraciones con otros sistemas (ver Figura **2-4**). A pesar que existen funcionalidades e integraciones por defecto algunas de estas características deben extenderse, personalizarse y/o reemplazarse, debido a los procesos internos que varían de cliente a cliente. En este punto, es donde los Partners ayudan a los clientes en la implementación de estos requisitos.

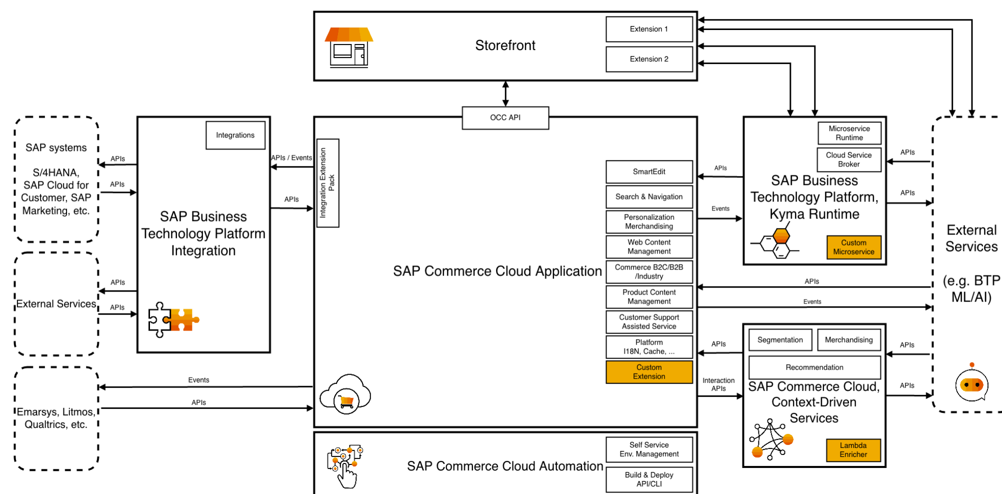

Figura **2-4**.: Diagrama de Arquitectura de integraciones SAP Commerce Cloud (Ceron, 2022).

Al ser un producto tan heterogéneo con tantas funcionalidades, los clientes suelen tener inconvenientes en adoptar el producto mediante el uso de buenas prácticas de desarrollo. Es por esto que SAP ha hecho esfuerzos previamente para apoyar a clientes y Partners en la adopción del producto mediante análisis automático de código fuente usando la herramienta SonarQube, como se puede ver en el portal **CX Works** [4] (MacWilliam, 2022), y por ende motivar a clientes y partners a evaluar la calidad de sus implementaciones.

[3] "*Backoffice*" Módulo de administración de los datos mediante una interfaz gráfica para usuarios internos.

[4] CX Works es el portal oficial de SAP donde se documentan las buenas prácticas recomendadas por SAP para su suite de

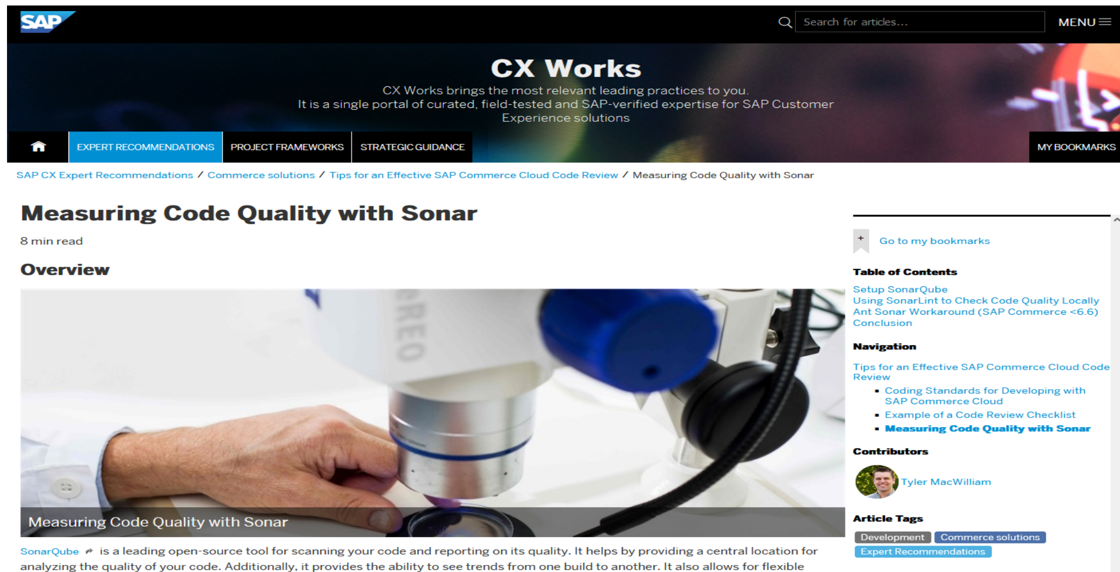

Figura **2-5**.: Artículo *Measuring Code with Sonar* del portal oficial de **CX Works**. (MacWilliam, 2022).

En el artículo *Measuring Code with Sonar* (MacWilliam, 2022) (Figura **2-5**), SAP documenta el paso a paso para ejecutar un análisis automático de código fuente sobre el código personalizado que ha escrito el Partner y/o el cliente. El artículo presenta un conjunto de reglas en el archivo de configuración `java-hybris-profile.xml`, el cual se importa directamente dentro de SonarQube para su posterior análisis. Esta guía también explica cómo configurar una instancia de SonarQube, la cual puede ser ejecutada localmente o mediante un sistema de integración continua.

Esta contribución fue documentada por el equipo de consultores expertos de SAP, y tiene el propósito de apoyar a los Partners y clientes, en la adopción del producto mediante el uso de buenas prácticas desde la perspectiva de la implementación. No obstante, este aporte no analiza el proceso de transición a la nube pública de Microsoft Azure, mediante el modelo de Plataforma como un servicio. Por lo tanto, el presente Trabajo Final de Maestría desea aportar en este sentido.

productos de Customer Experience (CX).

2.4. Selección de módulos de SAP Commerce y herramientas para el análisis automático de código fuente

SAP Commerce es un producto con una cantidad de funcionalidades muy amplia como se puede evidenciar en el diagrama de arquitectura de la Figura **2-4**, por lo tanto, es necesario delimitar el alcance de este trabajo por motivos de tiempo y relevancia. Esta sección define los criterios para la selección de módulos de SAP Commerce (Sección 2.4.1) y su selección con base en ellos (Sección 2.4.2). Por último, en la Sección 2.4.3 se presenta un análisis comparativo de herramientas para el análisis estático de código fuente en los módulos seleccionados de SAP Commerce.

2.4.1. Criterios de selección de módulos de SAP Commerce

SAP Commerce es una herramienta empresarial robusta que posee muchos módulos con funcionalidades para distintos propósitos. Algunos de estos componentes de SAP Commerce son desarrollados y mantenidos por la compañía SAP, y algunos otros elementos de la arquitectura no son parte de la solución pero se integran con ella. Por lo tanto, se propone el diagrama de decisión de la Figura **2-6**, junto con los criterios usados para definir los módulos que se analizarán en el presente proyecto.
En la Figura **2-6** se presenta el diagrama de flujo de decisión y en la siguiente lista se explica en detalle cada una de sus tareas:

1. **Lista módulos de SAP Commerce**: Teniendo en cuenta los componentes presentados en los diagramas de arquitectura de las Figuras **2-3** y **2-4** respectivamente, se listan para ser analizados uno a uno por el flujo.

2. **Para cada módulo listado**: Las tareas y bifurcaciones posteriores se realizarán para cada módulo listado en la tarea 1.

3. **¿Directamente relacionado con SAP Commerce?**: Evalúa si ese módulo pertenece o no a la solución SAP Commerce.

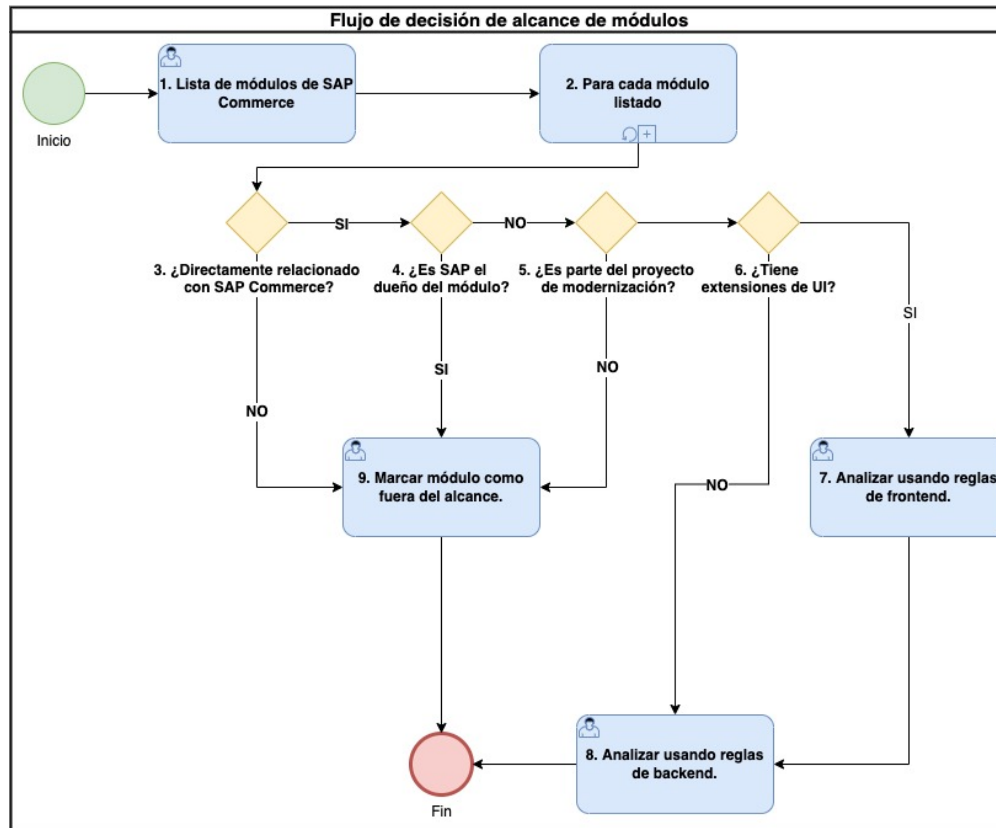

Figura **2-6**.: Diagrama de decisión para selección de módulos de SAP Commerce.

a) **SI**: El módulo pertenece directamente a SAP Commerce por lo tanto se debe evaluar el paso 4.

b) **NO**: A pesar de que el componente está dentro de la arquitectura, el módulo o componente no hace parte de la solución SAP Commerce; por lo tanto, se debe ir a la tarea 9.

4. **¿Es SAP el dueño del módulo?**: Evalúa si ese módulo es mantenido y desarrollado por el fabricante SAP.

   a) **SI**: Este módulo se encuentra fuera del alcance ya que el fabricante garantiza que funcionará correctamente en la nube o proveerá una alternativa a seguir. Ir a la tarea 9.

   b) **NO**: Este módulo no hace parte de las funcionalidades por defecto, por lo tanto pertenece al Partner o cliente. Continuar a la pregunta 5.

5. **¿Es parte del proyecto de modernización?**: Evalúa si es necesario que este módulo haga parte

del proyecto de modernización a la nube.

a) **SI**: Al ser necesaria su migración debe ser analizado. Continuar con la pregunta 6.

b) **NO**: El módulo no hace parte del alcance ya que no es soportado por la nueva infraestructura o está en proceso de desuso. Ir a la tarea 9.

6. **¿Tiene extensiones de UI?**: Evalúa si el módulo posee extensiones que tienen un componente de interfaz gráfica.

    a) **SI**: Debe ser analizado usando reglas de front-end. Continuar con la tarea 7.

    b) **NO**: No es necesario analizar herramientas de la capa de presentación. Continuar con la tarea 8.

7. **Analizar usando reglas de front-end**: El módulo usa tecnologías de la capa de presentación por lo tanto debe analizar componentes de UI.

8. **Analizar usando reglas de back-end**: Todas las extensiones son construidas sobre Spring Framework (Spring, 2020) por lo tanto todas deben ser analizadas por reglas correspondientes a Spring.

9. **Marcar módulo como fuera del alcance**: El módulo no será analizado por la estrategia de análisis automático de código toda vez que no hace parte integral de la solución SAP Commerce.

2.4.2. Selección de módulos de SAP Commerce

Esta sección se encarga de listar los módulos que son parte de SAP Commerce teniendo en cuenta la documentación oficial del producto (Ceron, 2022). En la Figura **2-7** se pueden observar mediante una paleta de colores los módulos de SAP Commerce alineados con el diagrama de arquitectura presentado en la Figura **2-4**. En azul se encuentran los módulos que son candidatos a ser incluidos dentro del alcance del análisis automático de código fuente, debido a que son personalizados por los clientes y necesitan de una revisión para la validación de las buenas prácticas. Adicionalmente, los módulos en morado y amarillo son administrados por SAP o productos independientes respectivamente, los cuales no es necesario analizarlos porque la empresa SAP da el soporte y compatibilidad.

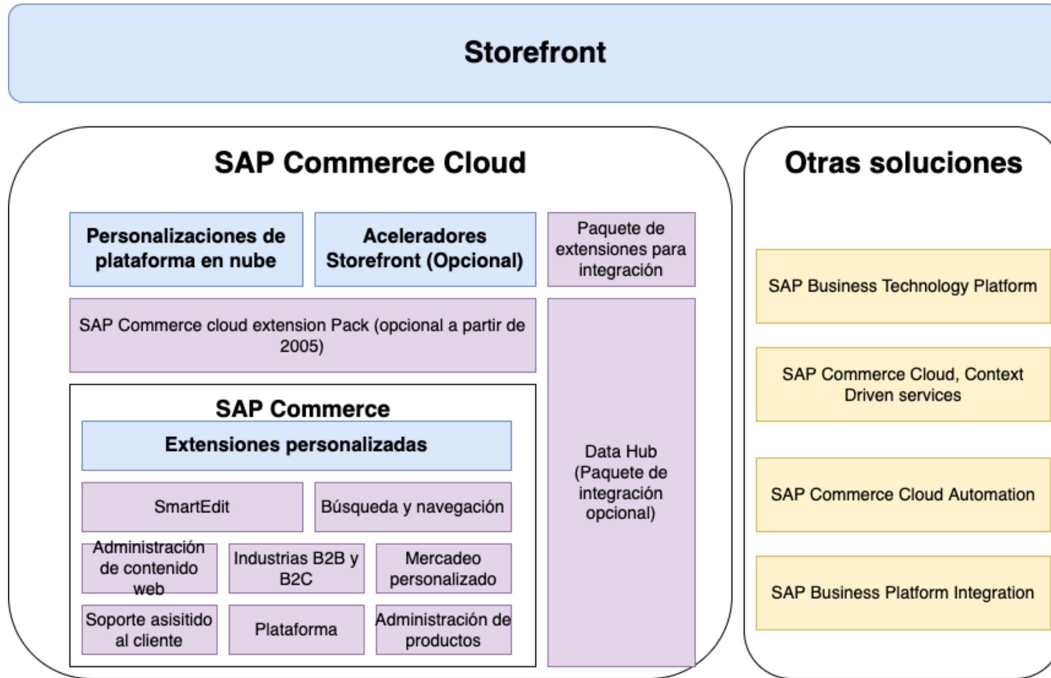

Figura **2-7**.: Diagrama de módulos definidos dentro del alcance del trabajo final.
Azul: módulos candidatos a ser incluidos dentro del alcance del análisis.
Morado: Módulos mantenidos y administrados por SAP los cuales están fuera del alcance.
Amarillo: Productos independientes que se integran con SAP Commerce y por lo tanto también se encuentran fuera del alcance.

A continuación, se lista cada uno de los módulos, especificando el nombre del módulo, su descripción, la decisión si serán o no analizados, y la justificación por la cual los módulos son candidatos para incluirlos dentro del alcance.

2.4.2.1. Storefront (Aceleradores):

Se analizará: Si.

Descripción: Este módulo se encarga de definir la capa de presentación web que utilizan los compradores para consultar y comprar productos y/o servicios en la tienda. Este módulo usualmente es personalizado ya que cada cliente define su propia imagen y funcionalidad.

Justificación: Ya que el módulo es personalizado por y/o para los clientes, es necesario hacer una inspección de código fuente para determinar si está acorde a las buenas prácticas para la migración. La

tarea 7 del flujo de selección se aplica para este caso.

2.4.2.2. Custom cloud extensions (Extensiones de nube personalizadas):

Se analizará: **Si**.

Descripción: Este módulo agrupa las extensiones de nube personalizadas que definen la estructura del proyecto que se desplegará en la nube, es decir los servicios web, las extensiones personalizadas, los servicios por defecto que se usarán en el producto y demás características que definen el despliegue de SAP Commerce en la nube de Microsoft Azure.

Justificación: Ya que el módulo es personalizado por y/o para los clientes, es necesario hacer una inspección de código fuente para determinar si está acorde a las buenas prácticas para la migración. Las tareas 7 y 8 del flujo de selección se aplican para este caso.

2.4.2.3. Custom extensions (Extensiones personalizadas):

Se analizará: **Si**.

Descripción: Este módulo agrupa las extensiones que se encargan de personalizar la lógica de negocio, integraciones con aplicaciones de terceros, acceso a la capa de datos, entre otros. SAP Commerce tiene un conjunto de funcionalidades de caja (por defecto) que cubren los casos de uso comunes de una tienda en línea. No obstante, algunos clientes tienen procesos de negocio distintos, por lo tanto, es necesario implementar estas características en SAP Commerce de forma personalizada.

Justificación: Ya que el módulo es personalizado por y/o para los clientes, es necesario hacer una inspección de código fuente para determinar si es acorde a las buenas prácticas para la migración. Las tareas 7 y 8 del flujo de selección se aplican para este caso.

2.4.2.4. OCC API (Servicios de integración vía REST):

Se analizará: **No**.

Descripción: La API Omni Commerce Connect (OCC) expone un amplio conjunto de servicios comerciales y de datos. Le permite integrar la funcionalidad de SAP Commerce Cloud en cualquier lugar de su entorno de aplicaciones.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 9 del flujo marca este módulo como fuera del alcance.

2.4.2.5. SmartEdit:

Se analizará: **No**.

Descripción: SAP Commerce SmartEdit permite a los administradores de contenido crear y administrar fácilmente el contenido de su sitio web sobre la marcha en diferentes puntos de inflexión y ponerlo a disposición de sus clientes con solo hacer clic en un botón.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 9 del flujo marca este módulo como fuera del alcance.

2.4.2.6. Búsqueda y navegación:

Se analizará: **No**.

Descripción: El módulo de búsqueda y navegación proporciona una variedad de funciones relacionadas con la creación y administración de configuraciones de búsqueda. Por ejemplo, puede crear plantillas de búsqueda administrables y personalizables.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 9 del flujo marca este módulo como fuera del alcance.

2.4.2.7. Personalization merchandising:

Se analizará: **No**.

Descripción: El módulo de mercadeo basado en el contexto proporciona una gama de características relacionadas con la definición de la configuración de integración. Por ejemplo, le permite definir los servicios de la plataforma en la nube y la sincronización del catálogo de productos.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es nece-

sario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 9 del flujo marca este módulo como fuera del alcance.

2.4.2.8. Gestor de contenido web (WCMS):

Se analizará: **No**.

Descripción: El módulo de gestión de contenido web (WCMS) proporciona un sistema de publicación multicanal que permite administrar fácilmente sitios web, incluidas sus páginas transaccionales y no transaccionales. Esta característica también permite a los usuarios administrar el contenido de forma centralizada, lo que facilita la actualización y el mantenimiento del contenido web.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 4 del flujo marca este módulo como fuera del alcance.

2.4.2.9. Commerce B2C/B2B/Industry:

Se analizará: **No**.

Descripción: El módulo de comercio B2B de SAP Commerce Cloud proporciona a las organizaciones B2B una solución multicanal flexible que se puede personalizar fácilmente para cumplir con los requisitos específicos del ciclo de compras B2B. Se pueden administrar múltiples modelos comerciales, canales y mercados en una sola plataforma. Por otra parte el módulo B2C proporciona la función de administración de productos y servicios que permiten soportar el modelo de negocio de venta directa a compradores al detal.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 4 del flujo marca este módulo como fuera del alcance.

2.4.2.10. Administrador de contenido de producto:

Se analizará: **No**.

Descripción: Backoffice Product Content Management es una herramienta de diseño centrada en el usuario que le permite administrar los datos de sus productos.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 4 del flujo marca este módulo como fuera del alcance.

2.4.2.11. Módulo de servicio asistido:

Se analizará: **No**.

Descripción: El módulo de servicio asistido proporciona características que permiten al personal de soporte de servicio usar la misma vista de escaparate que usan sus clientes y brindar un servicio superior. Por ejemplo, el agente de servicio al cliente puede crear una nueva cuenta de cliente en nombre del cliente.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 4 del flujo marca este módulo como fuera del alcance.

2.4.2.12. Plataforma:

Se analizará: **No**.

Descripción: La plataforma proporciona una gama de funciones relacionadas con la funcionalidad principal de una instalación de SAP Commerce Cloud, es decir, contenedorización, gestión de datos, sincronización, seguridad o localización. Es considerado como el módulo base y **estrictamente requerido** en SAP Commerce, el cual es utilizado directa o indirectamente por todos los demás módulos.

Justificación: Este módulo es desarrollado y mantenido por la empresa SAP, por lo tanto no es necesario realizar un análisis de código. La compañía garantiza el uso de las buenas prácticas y la compatibilidad del módulo en la nube. La tarea 4 del flujo marca este módulo como fuera del alcance.

2.4.2.13. SAP Commerce cloud automation:

Se analizará: **No**.

Descripción: Este producto provee la implementación de las características de Plataforma como un servicio para que SAP Commerce pueda ser desplegado en la infraestructura de Microsoft Azure. Provee características de automatización de construcción y despliegue de código fuente, administración

de ambientes, entre otros.

Justificación: Este producto se integra con SAP Commerce, no obstante, es distinto y no hace parte de la solución de comercio en línea. Por lo tanto, la actividad 3 remueve este producto del alcance de este trabajo.

2.4.2.14. SAP Commerce cloud intelligent selling services:

Se analizará: **No**.

Descripción: Los servicios de venta inteligente brindan personalización en tiempo real a través de carruseles de productos de recomendaciones y comercialización. También proporciona herramientas de prueba, generación de informes y administración para comprender, monitorear y mejorar el rendimiento de los productos en el mercado.

Justificación: Este producto se integra con SAP Commerce, no obstante, es distinto y no hace parte de la solución de comercio en línea. Por lo tanto, la actividad 3 remueve este producto del alcance de este trabajo.

2.4.2.15. SAP Business Technology Platform:

Se analizará: **No**.

Descripción: SAP Business technology platform ofrece una forma integrar diferentes portafolios tecnológicos para cualquier negocio. Se enfoca en bases de datos y administración de datos, desarrollo e integración de aplicaciones, análisis y tecnologías inteligentes. Como resultado, es sencillo y posible convertir los datos en un negocio para obtener el mejor valor al mismo tiempo que se ayuda a promover los procesos comerciales de un extremo a otro. Además, los desarrolladores pueden convertir datos tanto en las instalaciones como en la nube.

Justificación: Este producto se integra con SAP Commerce, no obstante, es distinto y no hace parte de la solución de comercio en línea. Por lo tanto, la actividad 3 remueve este producto del alcance de este trabajo.

2.4.2.16. Servicios externos y otras soluciones:

Se analizará: No.

Descripción: Son soluciones externas que pertenecen al portafolio de *Experiencia de Cliente* de SAP. Estas soluciones crean un recorrido del cliente conectado basado en la empatía y la confianza con el portafolio de SAP Customer Experience para ayudar a las empresas a innovar, integrar y ser ágil en torno a soluciones basadas en los clientes.

Justificación: Estos productos se integra con SAP Commerce, no obstante, es distinto y no hace parte de la solución de comercio en línea. Por lo tanto, la actividad 3 remueve este producto del alcance de este trabajo.

2.4.3. Análisis comparativo de herramientas para el análisis estático de código fuente en SAP Commerce

En esta sección se hace un listado de herramientas existentes en el mercado que permiten hacer análisis estático de código fuente. Inicialmente se definen los criterios de evaluación de las herramientas que hacen parte del análisis, posteriormente se hace un comparativo de estas herramientas mediante el uso de una tabla.

Anteriormente, en la Sección 2.3.2 se hizo mención al trabajo previo de análisis automático de código fuente sobre SAP Commerce (MacWilliam, 2022). No obstante, esa implementación no comprende la validación de mejores prácticas sobre los cambios que se deben realizar al migrar la plataforma de infraestructura on-prem a la nube de Microsoft Azure.

A grandes rasgos SAP Commerce usa Spring Framework para el manejo de contexto de la capa de aplicación y acceso a datos, y Angular para la definición de la capa de presentación. Por lo tanto, la herramienta o estrategia planteada para hacer análisis automático de código fuente debe soportar los lenguajes Java, Javascript, Typescript, y JSON. Por ende las herramientas analizadas en este trabajo fueron preseleccionadas teniendo en cuenta tales lenguajes.

La Tabla **2-2** presenta el análisis comparativo de herramientas candidatas para el análisis de código fuente en SAP Commerce, teniendo en cuenta los siguientes criterios:

- **Extensibilidad:** define si la herramienta es extensible con el fin de definir reglas o una imple-

mentación personalizadas para el análisis automático de código fuente. *Código de colores:* La herramienta se puede extender con funcionalidades personalizadas, la herramienta no se puede extender.

- **Última versión:** indicador de la versión actual y su fecha de lanzamiento, esto, con el fin de determinar si la herramienta está siendo soportada o no. *Código de colores*: La última versión del producto tiene menos de un año, la última versión del producto tiene más de un año.

- **Licencia:** Indica el tipo de licencia de código abierto, código cerrado y/o comercial. Este proyecto se ejecuta bajo el marco de un producto comercial de código cerrado, no obstante, la estrategia planteada podrá ser usada o extendida de forma libre bajo el modelo de licenciamiento GPL. *Código de colores*: La licencia es open source, licenciamiento comercial con versión gratuita, la licencia es de código cerrado de distribución comercial.

- **Salida:** tipo de formato en el que se produce el reporte de hallazgos encontrados por la herramienta.

Tabla **2-2**.: Comparativo herramientas candidatas para análisis de código fuente en SAP Commerce

#	Herramienta	Extensible	Última versión	Licencia	Salida
1	**ANTLR** [5]	Si	4.10.1, Abril 2022	Open source	Lista
2	**Axivion Bauhaus suite**	No	7.4, Junio 2022	Comercial	Lista
3	**CAST**	No	5.4.31, Julio 2022	Comercial	Lista
4	**Checkmarx**	No	9.4.0, Enero 2022	Comercial	Lista
5	**Codacy**	No	Cloud July 2022	Comercial	Lista y grafos visuales
6	**Coverity**	No	Cloud July 2022	Freemium, open source	Lista y grafos visuales
7	**Infer**	Si	1.1.0, Marzo 2021	Open source	Lista
8	**GrammaTech**	No	6.2, Diciembre 2021	Comercial	Lista
9	**Java Parser**	Si	3.24.3, Julio 2022	Open source	Lista

[5] Si bien ANTLR no es un analizador estático de código, si permite definir reglas basadas en árboles de sintaxis abstractos que permiten hacer validaciones.

#	Herramienta	Extensible	Última versión	Licencia	Salida
10	**Kiuwan**	No	Julio de 2022	Comercial	Lista
11	**Klocwork**	No	2022.2, Julio 2022	Comercial	Lista
12	**Microfocus Fortify**	No	20.2.4, Feb 2022	Comercial	Lista
13	**PMD**	Si	6.48.0, Julio 2022	Open source	Lista
14	**SonarQube**	Si	8.9.9 LTS, Junio 2022	Versión Comunitaria y de pago. Código cerrado	HTML
15	**Squale**	Si	7.1, Mayo 2011	Open source	Corrección de código
16	**Yasca**	Si	2.2, Mayo 2010	Open source	HTML, XML

Fuente: Elaboración propia y adaptación de (Ashfaq et al., 2019) y (DeepCode et al., 2022)

Después de considerar las características presentadas en la Tabla **2-2**, se tomó la decisión de seleccionar PMD como la herramienta principal para el análisis automático de código fuente en SAP Commerce. Esta elección se basó en las siguientes razones:

En primer lugar, PMD es una herramienta de software libre y de código abierto, lo que significa que su uso no está restringido por cuestiones comerciales. Esto permite aprovechar sus funcionalidades sin limitaciones y adaptarlo según las necesidades específicas del proyecto.

Además, al ser el código fuente de PMD abierto, es posible acceder y comprender en detalle la ejecución del análisis. Esto brinda la oportunidad de realizar personalizaciones y ajustes necesarios para adaptar el análisis automático de código fuente a las particularidades de SAP Commerce.

Otro aspecto relevante es que PMD utiliza librerías como ANTLR en su núcleo. Esto proporciona la capacidad de ampliar su funcionalidad para soportar lenguajes de dominio específico mediante la colaboración de la comunidad de software libre. Además, ANTLR ofrece una gramática flexible que facilita la creación de reglas y patrones de análisis personalizados.

3. Estrategia de análisis automático de código fuente en SAP Commerce

Este capítulo explica con detalle la estrategia propuesta para hacer análisis automático de código fuente en SAP Commerce. En la Figura **3-1** se representan las 3 fases de la estrategia:

1. **Análisis de características incompatibles con SAP Commerce:** Sección 3.1.
    a) **Análisis de extensiones obsoletas:** Sub-sección 3.1.1.
    b) **Incompatibilidades de SAP Commerce con arquitectura cloud:** Sub-sección 3.1.2.
    c) **Detección de librerías de terceros:** Sub-sección 3.1.3.
    d) **Detección de palabras clave de motores de base de datos:** Sub-sección 3.1.4.
2. **Análisis de código Java:** Sección 3.2.
3. **Análisis de código Javascript:** Sección 3.3.

3.1. Análisis de características incompatibles con SAP Commerce

SAP Commerce rediseñó su arquitectura con el fin de adaptar la solución a tecnologías basadas en la nube, por lo tanto, algunas de las funcionalidades de SAP Commerce son incompatibles con SAP Commerce Cloud. Para detectar estas características incompatibles el análisis se divide en 4 fases que se describen a continuación.

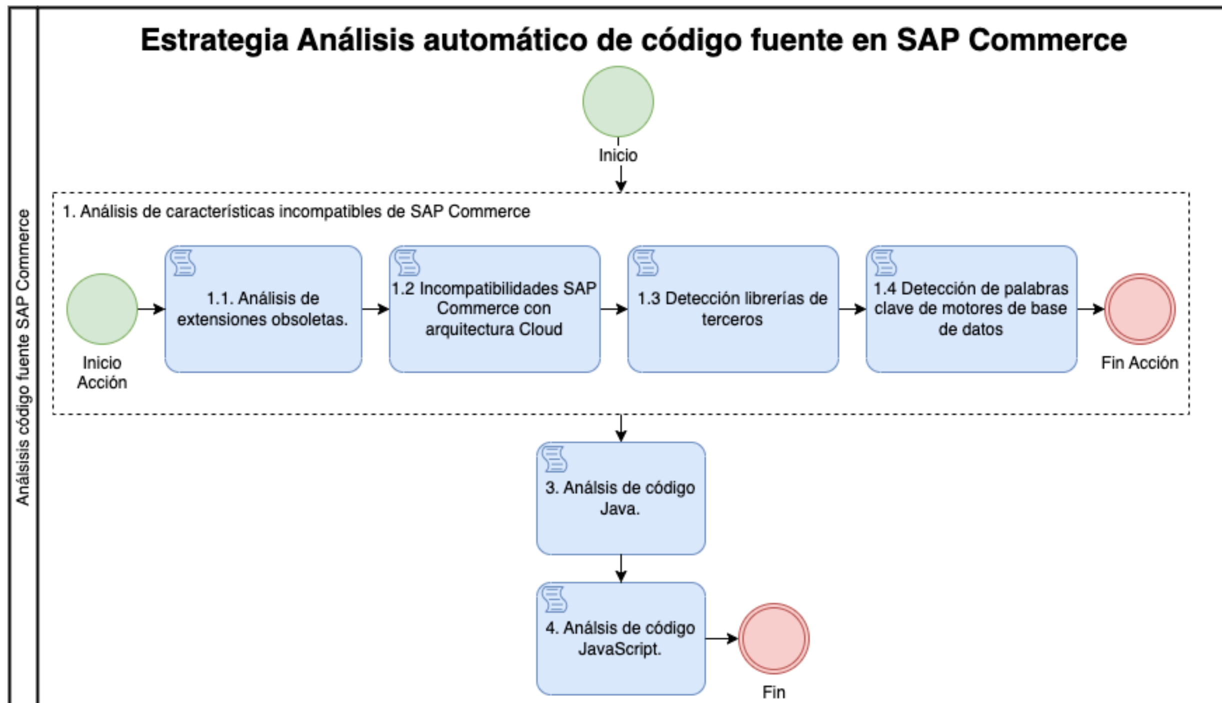

Figura **3-1**.: Diagrama de procesos de la estrategia de análisis automático de código fuente en SAP Commerce.

3.1.1. Detección de uso de extensiones obsoletas

El uso de extensiones obsoletas no es permitido en la infraestructura de SAP Commerce en la nube, por lo tanto, esta regla valida que no se haga referencia a extensiones obsoletas, ya sea en el archivo de extensiones `localextensions.xml` y/o en el archivo de dependencias de cada extensión llamado `extensioninfo.xml`.

La documentación oficial de SAP especifica el listado de extensiones obsoletas a la fecha de escritura del documento: 20 de octubre de 2023.

3.1.1.1. Detección de extensiones obsoletas en el archivo de extensiones

El archivo `localextensions.xml` contiene la lista de extensiones que se incluirán en tiempo de compilación y ejecución de SAP Commerce. Con el fin de extender la funcionalidad de SAP Commerce, se deben agregar extensiones a esta lista (SAP Local extensions, 2023).

Este archivo contiene una combinación de las extensiones por defecto de SAP Commerce junto con

extensiones personalizadas. En el bloque de código 3.1 se encuentra el ejemplo de la regla aplicada para este archivo.

```
1 <hybrisconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2     xsi:noNamespaceSchemaLocation="resources/schemas/extensions.xsd">
3   <extensions>
4     <!-- Incorrecto -->
5     <extension name="acceleratorstorefrontcommons"/>
6   </extensions>
7 </hybrisconfig>
```

Bloque de código 3.1: Detección del uso de extensiones obsoletas en el archivo localextensions.xml

El valor del atributo `name` en la línea 5 del bloque de código 3.1 no puede contener dependencia a ninguna extensión obsoleta.

3.1.1.2. Detección de extensiones obsoletas en el archivo de información de extensión

El archivo `extensioninfo.xml` está presente en cada extensión, y permite configurar sus características y dependencias. Este archivo es usado por el sistema de compilación de SAP Commerce para determinar de cuáles otras extensiones depende para un correcto funcionamiento. Este archivo puede contener una combinación de las extensiones por defecto de SAP Commerce junto con extensiones personalizadas. En el bloque de código 3.2 se encuentra el ejemplo de la regla aplicada para este archivo.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <extensioninfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="extensioninfo.xsd">
3     <extension abstractclassprefix="Generated" classprefix="Concerttours"
          jaloLogicFree="true" managername="ConcerttoursManager"
          managersuperclass="de.hybris.platform.jalo.extension.Extension" name="
          concerttours" usemaven="false">
4         <!-- you should add all required extensions to this list, except
              platform extensions which are automatically required -->
5         <!-- Incorrecto -->
```

```
6         <requires-extension name="acceleratorstorefrontcommons"/>
7         <coremodule generated="true" manager="concerttours.jalo.
              ConcerttoursManager" packageroot="concerttours"/>
8         <webmodule jspcompile="false" webroot="/concerttours"/>
9     </extension>
10 </extensioninfo>
```

Bloque de código 3.2: Detección del uso de extensiones obsoletas en los archivos extensioninfo.xml

El valor del atributo `name` de la línea 6 del bloque de código 3.2 no puede tener una dependencia a ninguna extensión obsoleta.

3.1.2. Incompatibilidades de SAP Commerce con arquitectura cloud

SAP Commerce posee configuraciones y características relacionadas con la definición de la infraestructura tecnológica. No obstante, estas características en el modelo de nube (cloud) son administradas directamente por SAP. Por lo tanto, estos ajustes no son necesarios en la arquitectura en la nube. Este análisis pretende detectar estas configuraciones y personalizaciones, que son incompatibles en la arquitectura en la nube, debido a que causan problemas de compilación. En esta sección se describen las características incompatibles: propiedades incompatibles para SAP Commerce Cloud (Sección 3.1.2.1) y propiedades potencialmente sensibles (Sección 3.1.2.2).

3.1.2.1. Propiedades incompatibles para SAP Commerce Cloud

Durante el proceso de construcción, las propiedades descritas a continuación son manejadas por SAP Commerce (SAP Managed Properties, 2023). Si son sobreescritas por el código fuente, generan problemas en la compilación y podría no compilar.
Esta regla se encarga de validar que ninguna de estas propiedades esté presente en la implementación de SAP Commerce (bloque de código 3.3).

```
1 // Conexión de base de datos.
2 db.url
3 db.driver
4 db.username
5 db.password
```

```
6 db.tableprefix
7 // Archivos de medios.
8 media.read.dir
9 media.replication.dirs
10 mediaweb.webroot
11 media.globalSettings.cloudAzureBlobStorageStrategy.connection
12 media.globalSettings.cloudAzureBlobStorageStrategy.public.base.url
13 // Administración de clúster
14 clustermode
15 cluster.id
16 cluster.maxid
17 cluster.broadcast.methods
18 cluster.broadcast.method.udp.multicastaddress
19 cluster.broadcast.method.udp.port
20 // Dynatrace - Solución de monitoreo.
21 dynatrace.enabled
22 dynatrace.agentlib
23 dynatrace.name
24 dynatrace.server
25 tomcat.generaloptions.dynatrace
26 // JMX
27 tomcat.generaloptions.jmxsettings
28 tomcat.jmx.port
29 tomcat.jmx.server.port
30 // Puertos de conexión
31 tomcat.http.port
32 tomcat.ssl.port
33 tomcat.ajp.port
34 tomcat.ajp.secureport
35 proxy.http.port
36 proxy.ssl.port
37 // JVM
38 tomcat.generaloptions
39 java.mem
40 tomcat.generaloptions.jmxsettings
```

```
41 tomcat.generaloptions.jvmsettings
42 tomcat.generaloptions.dynatrace
43 tomcat.generaloptions.GC
44 standalone.javaoptions
45 // Niveles de registro de Logs
46 log4j.threshold
47 // Tenants
48 installed.tenants
49 tenant.restart.on.connection.error
50 // Caching
51 regionalcache.entityregion.evictionpolicy
52 regioncache.stats.enabled
53 cms.cache.enabled
54 regioncache.entityregion.size
55 // Storefront
56 storefront.btg.enabled
57 storefront.resourceBundle.cacheSeconds
58 showStorefrontDebugInfo
59 storefront.show.debug.info
60 storefront.granule.enabled
61 storefront.staticResourceFilter.response.header.Cache-Control
62 addonfilter.active
63 default.session.timeout
64 // Solr
65 solrserver.instances.default.autostart
66 // DataHub
67 datahub.security.https.enabled (default value = false)
68 // Nodos de backoffice
69 spring.session.enabled
70 spring.session.hac.save
71 backofficesearch.cronjob.nodegroup
72 spring.session.hac.cookie.name
73 spring.session.hac.cookie.path
74 task.engine.exclusive.mode
75 cluster.node.groups
```

```
76 // Otras
77 multicountrysampledataaddon.import.active
78 bootstrap.init.type.system.custom.indices.use.items.definitions (default value
       = true)
79 bootstrap.init.type.system.custom.index.ignore.names.starting.with (default
      value = nci_wi_, idx_*dba)
```

Bloque de código 3.3: Propiedades de SAP Commerce administradas por SAP (SAP Managed Properties, 2023).

3.1.2.2. Propiedades potencialmente sensibles

SAP recomienda que las propiedades sensibles como contraseñas y secretos no se encuentre en el repositorio de código por motivos de seguridad. SAP Commerce cloud recomienda que estas propiedades sensibles se manejen mediante archivos estáticos que se deben almacenar y cargar en SAP Commerce en la nube de forma segura (SAP Sensitive configuration, 2023).

Esta regla se encarga de buscar en los archivos de propiedades de Java `*.properties` que no se encuentren las palabras clave: `secret`, `password`, `key`, `pass`, y así identificar el incorrecto manejo de las propiedades sensibles en el código fuente (bloque de código 3.4).

```
1 // Incorrecto. Los archivos de propiedades no deben contener propiedades
      sensibles.
2 property.sensitive.secret=secret
3 property.sensitive.password=password
4 property.sensitive.key=key
5 property.sensitive.pass=pass
```

Bloque de código 3.4: Detección de propiedades potencialmente sensibles.

3.1.3. Detección de librerías de terceros

Es habitual que en el desarrollo de aplicaciones los desarrolladores usen librerías de Java desarrolladas por terceros. SAP recomienda que siempre se use la última versión disponible de las librerías, toda vez, que estas últimas versiones contienen mejoras de rendimiento, corrección de defectos y vulnerabilidades.

Si bien el uso de las últimas versiones no garantiza el inexistencia de defectos o vulnerabilidades, si se pueden evitar problemas como la conocida brecha de seguridad de la librería log4j, cuya resolución fue la de actualizar a la última versión de la librería (Park and Lee, 2023).
El objetivo de esta regla es el de generar un reporte con las librerías usadas por la implementación personalizada de SAP Commerce, así como se puede ver en la Tabla **3-1**. Este reporte busca llamar la atención con respecto a las siguientes características:

- **Extensión:** Nombre de la extensión que está haciendo uso de la librería de terceros.
- **Librería:** Nombre de la librería.
- **Ubicación relativa:** Carpeta presente en el sistema de archivos que contiene la librería de java.
- **Versión:** Versión actual de la librería.
- **Última versión:** Última versión encontrada en el repositorio **público de maven**. Si la librería no se encuentra en este repositorio, no se considerará dentro del análisis.

Nótese que en la Tabla **3-1** la librería json-taglib se encuentra en la última versión, no obstante, se incluyó dentro de la tabla con fines ilustrativos más no se incluirá en el reporte programático. Lo anterior debido a que una librería que se encuentre en la última versión no incurre en ningún tipo de violación.

Tabla **3-1**.: Reporte de versiones de librerías de terceros

Extensión	Librería	Ubicación relativa	Versión	Última versión
extcore	dx-java	./lib/	1.2.0	1.5.2
extnotifications	json-taglib	./web/webroot/WEB-INF/lib/	0.4.1	0.4.1

Fuente: Elaboración propia.

3.1.4. Detección de palabras clave reservadas de motores de base de datos SQL

SAP Commerce posee dos lenguajes de dominio específico para el acceso y escritura de los datos: Impex, para importar y exportar datos en la base de datos; y Flexible search, el cual es un lenguaje similar a SQL que permite hacer búsqueda sobre las entidades de la base de datos. El objetivo de estos lenguajes de dominio específico es el de que SAP Commerce sea una herramienta agnóstica de la base de datos y se pueda usar el motor de base de datos de preferencia del cliente. No obstante, algunos desarrolladores usan funciones propias de los motores de base de datos, lo cual genera errores a la hora de migrar a la infraestructura en la nube, toda vez que el motor de base de datos es Azure SQL y este motor no tiene funciones propias de motores como Oracle, MySQL, SAP HANA y/o PostgreSQL. Es considerada mala práctica usar palabras clave y funciones propias del motor de base de datos SQL, ya que SAP Commerce soporta varias bases de datos SQL y no debe depender de una marca y/o tecnología específica.

La regla a continuación permite reportar aquellas clases que poseen funciones propias de los motores de base de datos, para que se puedan remover de las consultas y se pueda migrar SAP Commerce a la nube de Microsoft Azure. Teniendo en cuenta lo mostrado por el bloque de código 3.5, el objetivo es analizar la ocurrencia de las palabras clave listadas en la tabla **3-2**.

```
1  // Correcto
2  final String queryString = //
3          "SELECT {p:" + NewsModel.PK + "} "//
4              + "FROM {" + NewsModel._TYPECODE + " AS p} " //
5              + "WHERE {date} >= DATE \'" + theDay + "\' "//
6              + "AND {date} <= DATE \'" + theNextDay + "\'";
7  // Incorrecto
8  final String queryString = //
9          "SELECT {p:" + NewsModel.PK + "} "//
10             + "FROM {" + NewsModel._TYPECODE + " AS p} " //
11             + "WHERE {date} >= DATE \'" + theDay + "\' "//
12             + "AND {date} <= CURRENT_DATE";// CURRENT_DATE es una función
```

```
            exclusiva del motor de base de datos Oracle
```

Bloque de código 3.5: Regla Migración SAP Commerce: Detección de uso de funciones y palabras reservadas clave del motor de base de datos.

Tabla **3-2**.: Funciones propias de los motores de base de datos soportados por SAP Commerce.

Motor de base de datos	Funciones y palabras clave
Oracle	**Cadenas de texto:** ASCII, ASCIISTR, CHR, COMPOSE, CONCAT, CONVERT, DECOMPOSE, DUMP, INITCAP, INSTR, INSTR2, INSTR4, INSTRB, INSTRC, LENGTH, LENGTH2, LENGTH4, LENGTHB, LENGTHC, LOWER, LPAD, LTRIM, NCHR, REGEXP_INSTR, REGEXP_REPLACE, REGEXP_SUBSTR, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR, TRANSLATE, TRIM, UPPER, VSIZE. **Numéricas:** ABS, ACOS, ASIN, ATAN, ATAN2, AVG, BITAND, CEIL, COS, COSH, COUNT, EXP, FLOOR, GREATEST, LEAST, LN, LOG, MAX, MEDIAN, MIN, MOD, POWER, REGEXP_COUNT, REMAINDER, ROUND (numbers), ROWNUM, SIGN, SIN, SINH, SQRT, SUM, TAN, TANH, TRUNC (numbers) **Fecha/tiempo:** ADD_MONTHS, CURRENT_DATE, CURRENT_TIMESTAMP, DBTIMEZONE, EXTRACT, LAST_DAY, LOCALTIMESTAMP, MONTHS_BETWEEN, NEW_TIME, NEXT_DAY, ROUND (dates), SESSIONTIMEZONE, SYSDATE, SYSTIMESTAMP, TRUNC (dates), TZ_OFFSET. **Conversión:** BIN_TO_NUM, CAST, CHARTOROWID, FROM_TZ, HEXTORAW, NUMTODSINTERVAL, NUMTOYMINTERVAL, RAWTOHEX, TO_CHAR, TO_CLOB, TO_DATE, TO_DSINTERVAL, TO_LOB, TO_MULTI_BYTE, TO_NCLOB, TO_NUMBER, TO_SINGLE_BYTE, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL **Analíticas:** CORR, COVAR_POP, COVAR_SAMP, CUME_DIST, DENSE_RANK, FIRST_VALUE, LAG, LAST_VALUE, LEAD, LISTAGG, NTH_VALUE, RANK, STDDEV, VAR_POP, VAR_SAMP, VARIANCE. **Avanzadas:** BFILENAME, CARDINALITY, CASE, COALESCE, DECODE, EMPTY_BLOB, EMPTY_CLOB, GROUP_ID, LNNVL, NANVL, NULLIF, NVL, NVL2, SYS_CONTEXT, UID, USER, USERENV, SQLCODE, SQLERRM
MySQL	**Cadenas de texto:** ASCII, CHAR_LENGTH, CHARACTER_LENGTH, CONCAT, CONCAT_WS, FIELD, FIND_IN_SET, FORMAT, INSERT, INSTR, LCASE, LEFT, LENGTH, LOCATE, LOWER, LPAD, LTRIM, MID, POSITION, REPEAT, REPLACE, REVERSE, RIGHT, RPAD, RTRIM, SPACE, STRCMP, SUBSTR, SUBSTRING, SUBSTRING_INDEX, TRIM, UCASE, UPPER. **Numéricas:** ABS, ACOS, ASIN, ATAN, ATAN2, AVG, CEIL, CEILING, COS, COT, COUNT, DEGREES, DIV, EXP, FLOOR, GREATEST, LEAST, LN, LOG, LOG10, LOG2, MAX, MIN, MOD, PI, POW, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT. **Fechas/tiempo:** ADDDATE, ADDTIME, CURDATE, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURTIME, DATE, DATEDIFF, DATE_ADD, DATE_FORMAT, DATE_SUB, DAY, DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, EXTRACT, FROM_DAYS, HOUR, LAST_DAY, LOCALTIME, LOCALTIMESTAMP, MAKEDATE, MAKETIME, MICROSECOND, MINUTE, MONTH, MONTHNAME, NOW, PERIOD_ADD, PERIOD_DIFF, QUARTER, SECOND, SEC_TO_TIME, STR_TO_DATE, SUBDATE, SUBTIME, SYSDATE, TIME, TIME_FORMAT, TIME_TO_SEC, TIMEDIFF, TIMESTAMP, TO_DAYS, WEEK, WEEKDAY, WEEKOFYEAR, YEAR, YEARWEEK. **Avanzadas:** BIN, BINARY, CASE, CAST, COALESCE, CONNECTION_ID, CONV, CONVERT, CURRENT_USER, DATABASE, IF, IFNULL, ISNULL, LAST_INSERT_ID, NULLIF, SESSION_USER, SYSTEM_USER, USER, VERSION.
SQL Server	**Cadenas de texto:** ASCII, CHAR, CHARINDEX, CONCAT, Concat with +, CONCAT_WS, DATALENGTH, DIFFERENCE, FORMAT, LEFT, LEN, LOWER, LTRIM, NCHAR, PATINDEX, QUOTENAME, REPLACE, REPLICATE, REVERSE, RIGHT, RTRIM, SOUNDEX, SPACE, STR, STUFF, SUBSTRING, TRANSLATE, TRIM, UNICODE, UPPER. **Numéricas:** ABS, ACOS, ASIN, ATAN, ATN2, AVG, CEILING, COUNT, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, MAX, MIN, PI, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, SQUARE, SUM, TAN. **Fechas/tiempo:** CURRENT_TIMESTAMP, DATEADD, DATEDIFF, DATEFROMPARTS, DATENAME, DATEPART, DAY, GETDATE, GETUTCDATE, ISDATE, MONTH, SYSDATETIME, YEAR. **Avanzadas:** CAST, COALESCE, CONVERT, CURRENT_USER, IIF, ISNULL, ISNUMERIC, NULLIF, SESSION_USER, SESSIONPROPERTY, SYSTEM_USER, USER_NAME.

PostgreSQL	**Cadenas de texto:** btrim, char_length, character_length, concat with ||, initcap, length, lower, lpad, ltrim, position, repeat, replace, rpad, rtrim, strpos, substring, translate, trim, upper. **Numéricas:** abs, avg, ceil, ceiling, count, div, exp, floor, max, min, mod, power, random, round, setseed, sign, sqrt, sum, trunc. **Fecha/tiempo:** age, current_date, current_time, current_timestamp, date_part, extract, localtime, localtimestamp, now. **Conversión:** to_char, to_date, to_number, to_timestamp.
SAP HANA	**Cadenas de texto:** ASCII, CHAR, CONCAT, LCASE, LEFT, LENGTH, LOCATE, LOWER, NCHAR, REPLACE, RIGHT, UPPER, UCASE, LPAD, LTRIM, RTRIM, STRTOBIN, SUBSTR_AFTER, SUBSTR_BEFORE, SUBSTRING, TRIM, UNICODE, RPAD, BINTOSTR. **Numéricas:** ABS, ACOS, ASIN, ATAN, ATAN2, BINTOHEX, BITAND, BITCOUNT, BITNOT, BITOR, BITSET, BITUNSET, BITXOR, CEIL, COS, COSH, COT, EXP, FLOOR, HEXTOBIN, LN, LOG, MOD, POWER, RAND, ROUND, SIGN, SIN, SINH, SQRT, TAN, TANH, UMINUS. **Fecha/Tiempo:** CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_UTCDATE, CURRENT_UTCTIME, CURRENT_UTCTIMESTAMP, DAYOFMONTH, HOUR, YEAR, DAYOFYEAR, DAYNAME, DAYS_BETWEEN, EXTRACT, NANO100_BETWEEN, NEXT_DAY, NOW, QUARTER, SECOND, SECONDS_BETWEEN, UTCTOLOCAL, WEEK, WEEKDAY, WORKDAYS_BETWEEN, ISOWEEK, LAST_DAY, LOCALTOUTC, MINUTE, MONTH, MONTHNAME, ADD_DAYS, ADD_MONTHS, ADD_SECONDS, ADD_WORKDAYS. **Avanzadas:** CAST, TO_ALPHANUM, TO_REAL, TO_TIME, TO_CLOB, TO_BIGINT, TO_BINARY, TO_BLOB, TO_DATE, TO_DATS, TO_DECIMAL, TO_DOUBLE, TO_FIXEDCHAR, TO_INT, TO_INTEGER, TO_NCLOB, TO_NVARCHAR, TO_TIMESTAMP, TO_TINYINT, TO_VARCHAR, TO_SECONDDATE, TO_SMALLDECIMAL, TO_SMALLINT, Current_Schema, Session_User

Fuente: Elaboración propia

3.2. Análisis de código Java

SAP Commerce usa como tecnología principal Spring Framework (Spring, 2020) junto con Java para el desarrollo de la capa de aplicación y de acceso a datos. Por lo tanto, en esta sección se describen reglas de buenas prácticas 3.2.1, estilo de código 3.2.2, diseño 3.2.3, documentación 3.2.4, código propenso a errores 3.2.5, multi-hilo 3.2.6, rendimiento 3.2.7 y seguridad 3.2.8 establecidos por la comunidad (PMD, 2022).

3.2.1. Buenas prácticas de código Java

Reglas que refuerzan buenas prácticas generalmente aceptadas para Java (PMD, 2022).

Tabla **3-3**.: Conjunto de reglas de buenas prácticas de Java

Buenas prácticas de Java			
#	Regla	Descripción	Ejemplo

#	Regla	Descripción	Ejemplo
1	Evite el uso de clases abstractas sin métodos abstractos.	La clase abstracta no contiene ningún método abstracto. Una clase abstracta sugiere una implementación incompleta, que debe ser completada por subclases que implementen los métodos abstractos. Si la clase está destinada a usarse solo como clase base (no para instanciarse directamente), se puede proporcionar un constructor protegido para evitar la instanciación directa.	A.1
2	Generación de descriptor de acceso de clase.	La creación de instancias por medio de constructores privados desde fuera de la clase del constructor a menudo provoca la generación de un descriptor de acceso. Un método de fábrica, o la no privatización del constructor, puede eliminar esta situación. El archivo de clase generado es en realidad una interfaz. Da a la clase de acceso la capacidad de invocar un nuevo constructor de alcance de paquete oculto que toma la interfaz como un parámetro complementario. Esto convierte a un constructor privado en uno con alcance de paquete y es difícil de discernir. Nota: esta regla solo se ejecuta para Java 10 o inferior. Desde Java 11, se ha implementado JEP 181: control de acceso basado en nidos. Esto significa que en Java 11 y versiones posteriores, las clases de acceso ya no se generan.	A.2
3	Generación de descriptor de acceso de método.	Al acceder a campos/métodos privados de otra clase, el compilador de Java generará métodos de acceso con visibilidad privada del paquete. Esto agrega gastos generales y cuenta con el método dex en Android. Esta situación se puede evitar cambiando la visibilidad del campo/método de privado a paquete-privado. Nota: esta regla solo se ejecuta para Java 10 o inferior. Desde Java 11, se ha implementado JEP 181: control de acceso basado en nido. Esto significa que en Java 11 y versiones posteriores, las clases de acceso ya no se generan.	A.3
4	Generación de descriptor de acceso de método.	Los constructores y métodos que reciben matrices deben clonar objetos y almacenar la copia. Esto evita que los cambios futuros del usuario afecten la matriz original.	A.4
5	Evite el uso de la clase `MessageDigest`como un campo de clase.	Declarar una instancia de `MessageDigest` como un campo hace que esta instancia esté directamente disponible para múltiples subprocesos. Si es posible, se debe evitar compartir instancias de `MessageDigest`, ya que conduce a resultados erróneos si el acceso no se sincroniza correctamente. Simplemente cree una nueva instancia y utilícela localmente, donde la necesite. Crear una nueva instancia es más fácil que sincronizar el acceso a una instancia compartida.	A.5
6	Evite el uso de `printStackTrace`	Evite el uso de `printStackTrace();`, en su lugar use `Logger`.	A.6
7	Evite reasignar variables en la sentencia de `catch`.	Debe evitarse la reasignación de variables de excepción capturadas en una instrucción `catch` debido a: Si es necesario, se puede agregar fácilmente captura múltiple y el código aún se compilará. Siguiendo el principio de menor sorpresa, queremos asegurarnos de que una variable capturada en una instrucción `catch` sea siempre la que se arroja en un bloque `try`.	A.7
8	Evite reasignar variables en ciclos.	La reasignación de variables de ciclo puede generar errores difíciles de encontrar. Evitar o limitar cómo se pueden cambiar estas variables.	A.8
9	Evite la reasignación de parámetros.	No se recomienda reasignar valores a los parámetros de un método o constructor, ya que esto puede dificultar la comprensión del código. El código a menudo se lee con la suposición de que los valores de los parámetros no cambian y, por lo tanto, una asignación viola el principio de la menor sorpresa. Esto es especialmente un problema si el parámetro está documentado, p. en el javadoc del método y el nuevo contenido difiere del contenido original documentado. Utilice variables locales temporales en su lugar. Esto le permite asignar un nuevo nombre, lo que hace que el código sea más comprensible. Tenga en cuenta que esta regla considera tanto los métodos como los constructores. Si hay asignaciones múltiples para un parámetro formal, solo se informa la primera asignación.	A.9

#	Regla	Descripción	Ejemplo
10	Evite el uso de `StringBuffers/StringBuilders`.	`StringBuffers/StringBuilders` pueden crecer considerablemente y, por lo tanto, pueden convertirse en una fuente de fugas de memoria si se mantienen dentro de objetos con una vida útil prolongada.	A.10
11	Evite la codificación fija de IPs.	La aplicación con direcciones IP codificadas de forma fija pueden volverse imposible de desplegar en algunos casos. Es preferible extraer estas propiedades en archivos configurables.	A.11
12	Valide los resultados obtenidos por un `ResultSet`	Compruebe siempre los valores de retorno de los métodos de navegación (siguiente, anterior, primero, último) de un `ResultSet`. Si el valor devuelto es `false`, debe manejarse correctamente.	A.12
13	Evite el uso de constantes en interfaces	Las Interfaces son usadas para definir tipos, las constantes son detalles de implementación los cuales deberían estar en clases o enumeraciones..	A.13
14	El caso `default` debe estar de últimas en la sentencia de `switch`	Por convención el caso `default` debe estar ubicado en el último lugar, en la sentencia `switch`.	A.14
15	Evite la inicialización con doble llaves.	La inicialización de doble llave es un patrón para inicializar, por ejemplo, colecciones de forma concisa. Pero implícitamente genera un nuevo archivo `.class`, y el objeto tiene una fuerte referencia al objeto adjunto. Por esas razones, es preferible inicializar el objeto normalmente, aunque sea detallado. Esta regla cuenta cualquier clase anónima que solo tenga un único inicializador como una instancia de inicialización de doble llave. Actualmente no hay forma de averiguar si un método llamado en el inicializador no es accesible desde fuera de la clase anónima, y esos casos legítimos deben suprimirse por el momento.	A.15
16	Ciclo `for` podría ser reemplazado por `foreach`.	Los bloques que iteran sobre colecciones se pueden reemplazar de forma segura con la sintaxis `foreach` . La regla considera bucles sobre listas, arreglos e iteradores. Es seguro reemplazar un bucle si solo usa la variable de índice para acceder a un elemento de la lista o matriz, solo tiene una declaración de actualización y recorre todos los elementos de la lista o matriz de izquierda a derecha.	A.16
17	Evite el uso de más de una variable de control para ciclos `for`.	Tener más de una variable de control para un ciclo `for` hace que sea difícil de inferir cual es el rango de valores usado para la iteración. Por convención, use una única variable de control en los ciclos `for`.	A.17
18	Valide el uso de un nivel de `Log`.	Tener más de una variable de control para un ciclo En cualquier lugar donde se use un nivel de `Log` es necesario validar si el nivel de log está habilitado, de lo contrario evite la concatenación y/o creación de `Strings`. Una forma alternativa de validar los niveles de `log` es mediante el uso de parámetros, formateadores y/o registro perezoso mediante el uso de funciones lambda. Estas alternativas dependen de la herramienta usada para `logging`	A.18
19	Test de `Junit 4` deben usar la anotación `RunWith`	En JUnit 3 los test suites se indicaban mediante el método `suite`. Para JUnit 4 los test suite deben indicarse mediante la anotación `@RunWith(Suite.class)`	A.19
20	Test de `Junit` 4 deben usar la anotación `@After`	En JUnit 3 el método `tearDown()` era usado para limpiar los datos requeridos por cada test. En JUnit 4 se sugiere que se use la anotación `@After` para cada test. En JUnit 5 se introdujeron las anotaciones `@AfterEach` y `@AfterAll` para ejecutar métodos luego de cada y todos los test respectivamente.	A.20
21	Test de `Junit` 4 deben usar la anotación `@Before`	En JUnit 3 el método `setUp()` era usado para limpiar los datos requeridos por cada test. En JUnit 4 se sugiere que se use la anotación `@Before` para cada test. En JUnit 5 se introdujeron las anotaciones `@ABeforeEach` y `@BeforeAll` para ejecutar métodos luego de cada y todos los test respectivamente.	A.21

#	Regla	Descripción	Ejemplo
22	Test de `Junit` 4 deben usar la anotación `@Test`	En JUnit 3 se ejecutan todos los métodos que inician con la palabra `test`. En JUnit 4, sólo los métodos anotados con la anotación `@Test` son ejecutados. En JUnit 5, se introdujeron las siguientes anotaciones para la ejecución de los test: `@RepeatedTest`, `@TestFactory`, `@TestTemplate` o `@ParameterizedTest`.	A.22
23	Test de `Junit` 5 debe usar encapsulamiento de tipo `package`	En JUnit 5 los métodos con encapsulamiento de tipo `package` son ejecutados. En contraste con JUnit 4 el cual requiere que los métodos sean declarados de tipo público para poder ser ejecutados. Por lo tanto, es una buena práctica definir los métodos con encapsulamiento `package`.	A.23
24	Aserciones de `Junit` deben incluir un mensaje	Las aserciones de JUnit debe incluir un mensaje informativo. Use la versión de tres parámetros de `assertEquals()`.	A.24
25	Test de `Junit` posee demasiadas aserciones	Los test unitarios no deberían tener muchas aserciones. Tener más de 2 aserciones es indicativo que el test es más complejo de lo debido y/o pierde su característica de unicidad.	A.25
26	Test de `Junit` debe contener al menos una aserción	El test de JUnit debería contener al menos una aserción. Esto hace que los tests sean más robustos, y los mensajes de la aserción provee una idea clara de lo que el test está intentando probar.	A.26
27	Test de `Junit` debe contener la propiedad `expected` cuando se lanza una excepción	En JUnit 4 es esperado que se use la anotación `@Test(expected)` para denotar que los test deben lanzar excepciones.	A.27
28	Test de `Junit` debe contener la propiedad `expected` cuando se lanza una excepción	Los literales de tipo `String` deben ir primero en las comparaciones, así se evita el lanzamiento de un `NullPointerException` cuando el segundo parámetro es `null`. Es de notar que al cambiar la posición en los métodos `compareTo` y `compareToIgnoreCase` es necesario cambiar el signo de la comparación.	A.28
29	Bajo acoplamiento	El uso de tipos de implementación como `HashSet` o `ArrayList` como referencia de objetos, limita la posibilidad de cambiar por otra implementación en el futuro. Cuando sea posible prefiera el uso de las interfaces como `Set` o `List` respectivamente.	A.29
30	Evite el retorno de los arreglos internos	Exponer arreglos internos viola el principio de encapsulamiento, toda vez, que se pueden agregar o remover elementos fuera del objeto propietario del arreglo. Es más seguro retornar una copia del arreglo.	A.30
31	Ausencia de la anotación `Override`.	Anotar los métodos sobre-escritos con la anotación `@Override`, asegura que el método se sobre-escriba en el tiempo de compilación. Adicionalmente, agrega legibilidad al código para identificar fácilmente los métodos sobre-escritos.	A.31
32	Una declaración por línea.	Java permite la declaración de varias variables del mismo tipo en la misma línea. No obstante, podría generar código difícil de leer.	A.32
33	Preservar `Stack Trace`.	Lanzar una nueva excepción en un bloque `catch` sin pasar la excepción original, genera que la causa original se pierda, lo que lo hace más difícil de depurar y corregir.	A.33
34	Uso de constructor para tipos primitivos..	El uso de contenedores para tipos primitivos fue declarado obsoleto desde Java 9 y no deberían ser usados. Inclusive antes de Java 9, estos pueden ser remplazados por el uso del método estático `valueOf`. En el caso de la clase `Boolean` las constantes `Boolean.TRUE` y `Boolean.FALSE` deben usarse en vez de `Boolean.valueOf()`.	A.34
35	Reemplace el uso de `Enumeration` con `Iterador`.	Considere reemplazar el uso de `Enumeration` con `java.util.Iterator`.	A.35
36	Reemplace `Hashtable` con `Map`.	Reemplace el uso de `Hashtable` con `Map`, cuando la seguridad de multi-hilos no sea necesaria.	A.36
37	Reemplace `Vector` con `List`.	Reemplace el uso de `Vector` con `List`, cuando la seguridad de multi-hilos no sea necesaria.	A.37

#	Regla	Descripción	Ejemplo
38	Simplifique el uso de aserciones	Prefiera el uso de aserciones más específicas en lugar de sobre usar `assertTrue` o `assertFalse`. Esto genera mejores mensajes de error y hace que las aserciones sean más legibles.	A.38
39	Sentencias `switch` deben tener un caso de `default`.	Las sentencias `switch` deben ser exhaustivas, de tal forma controlar el flujo de ejecución. Esto se controla mediante la adición del caso `default`. Si el `switch` está basado en una enumeración, debe existir un caso por cada uno de los posibles valores de la misma.	A.39
40	Evite el uso de `System.(out|err).println`.	El uso de `System.(err|out).println()` es común para depurar errores, no obstante, en muchos casos estas banderas no son removidas. Reemplácelo por `logger` que ofrece características tales como: niveles de log y la posibilidad de habilitar y deshabilitarlo en ambientes productivos.	A.40
41	Evite el uso de `System.(out|err).println`.	Esta regla reporta asignaciones que no son usadas luego de que la variable ha sido reescrita. Se reportan los siguientes casos: 1. La variable no se lee luego de la asignación. 2. El valor asignado siempre es sobre-escrito por otra sentencia, sin antes haberse leído su valor. La regla no considera asignaciones echas a atributos de la clase que usan el prefijo `this`, ya que el valor puede ser leído desde otra clase externa.	A.41
42	Parámetro no leído.	Reporta parámetros de métodos privados que no son leídos. Debido a que remover parámetros de métodos públicos puede generar problemas, estos son ignorados por la regla.	A.42
43	Variable local no usada.	Reporta variables locales que no son usadas.	A.43
44	Atributo privado no usado.	Esta regla detecta cuando un atributo privado de una clase es declarado, se le asigna un valor, pero nunca es leída.	A.44
45	Método privado no usado.	Reporta los métodos privados que son declarados pero no llamados.	A.45
46	Use `Collections.isEmpty()`.	El método `isEmpty` de la interfaz `java.util.Collection` permite evaluar si una colección contiene o no elementos dentro de ella. Es preferible usar `isEmpty` a el uso de `size()== 0`	A.46
47	Use `StandardCharsets`	En Java 7 se introdujo la clase `StandardCharsets`, la cual provee constantes con los conjuntos de caracteres más comunes. Cuando sea posible se prefiere el uso de `StandardCharsets.UTF_8` sobre `Charset.forName("UTF-8")`.	A.47
48	Use `Try` con el manejo de recursos.	En Java 7 se introdujo la creación de sentencias `try` para el manejo de recursos que se deben cerrar, esto evita la necesidad de crear un bloque de tipo `finally`, dentro del cual se debía crear otro bloque `try/catch` para el correcto cierre de los recursos. Si una excepción ocurría dentro del segundo `try/catch` la excepción original se pierde, lo cual hace más complejo el proceso de depuración.	A.48
49	Use argumentos variables `varargs`.	En Java 5 se introdujo la opción de usar argumentos variables `varargs` para métodos y constructores. Esta característica permite manejar argumentos variables, sin la necesidad de crear un arreglo.	A.49
50	Ciclo `while` con literal booleano.	`do {} while (true);` Requiere leer todo el bloque para un ciclo infinito, prefiera el uso de `while (true){}`. `do {} while (false);` No es necesario usar un ciclo para ejecutarlo una vez. `while (false){}` nunca se va a ejecutar por lo tanto no es necesario ese bloque de código.	A.50

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.2. Estilo código Java

Reglas que imponen un estilo de codificación específico que sea legible (PMD, 2022).

Tabla **3-4**.: Conjunto de reglas Estilo de codificación para Java

#	Regla	Descripción	Ejemplo
1	Clases deben tener al menos un constructor.	Clases no estáticas deben tener al menos un constructor.	A.51
2	Evite signo de dinero $.	Evite el signo de dinero $ en el nombrado de variables, métodos, clases y/o interfaces.	A.52
3	Evite atributos protegidos en clases finales.	No use atributos protegidos en clases finales, toda vez que estas no se pueden extender. Prefiera el uso de variables privadas o de tipo paquete (package).	A.53
4	Evite métodos protegidos en clases finales.	No use métodos protegidos en clases finales, toda vez que estas no se pueden extender. Prefiera el uso de métodos privados o de tipo paquete (package).	A.54
5	Evite el uso de código Nativo (JNI).	El uso de JNI (*Java Native Interface*) reduce la portabilidad de la aplicación e incrementa considerablemente su tiempo de mantenimiento.	A.55
6	Evite el uso de *get* como prefijo para métodos que retornan booleanos.	Los métodos que retornan booleanos deben tener prefijos más coherentes como *is*, *has*, *can*, *will*, *should*.	A.56
7	Llame a `super` en el constructor para clases hijas.	Se considera buena práctica el llamar al constructor del padre mediante `super();` cuando una clase hereda de otra.	A.57
8	Convención de llamado de clases.	Use las siguientes expresiones regulares para el correcto nombrado de clases: **Clase regular:** `[A-Z][a-zA-Z0-9]*`. **Clase abstracta:** `[A-Z][a-zA-Z0-9]*`, **Interfaz:** `[A-Z][a-zA-Z0-9]*`, **Enumeración:** `[A-Z][a-zA-Z0-9]*`, **Anotación:** `[A-Z][a-zA-Z0-9]*` y **Test:** `^Test.*|^[A-Z][a-zA-Z0-9]*Test(s|Case)?`	A.58
9	Comente el modificador de acceso por defecto.	Para modificadores de acceso de tipo `package` el cual es el modificador por defecto, considere agregar un comentario `/* default */`.	A.59
10	Operador ternario confuso.	Evite el caso negativo en la expresión `if` y el caso positivo en la rama `else`, de esa manera el código será más fácil de leer y entender.	A.60
11	Use llaves en sentencias de control.	Use llaves en las siguientes sentencias de control: `while, for, do, if, else`.	A.61
12	Sentencias de control vacías.	Esta regla reporta aquellas sentencias de control que contienen un cuerpo vacío, así como inicializadores vacíos. Se analizan las sentencias de control: `try, finally, switch, syncronized, if, while, for, do, {}`.	A.62
13	Métodos vacíos en clases abstractas deben ser abstractos.	Esta regla reporta aquellas sentencias de control que contienen un cuerpo vacío, así como inicializadores vacíos. Se analizan las sentencias de control: `try, finally, switch, syncronized, if, while, for, do, {}`.	A.63
14	Extensión de `Object`	No es necesario extender de forma explícita el objeto `Object`.	A.64
15	Declaración de atributos de clase deben estar al inicio.	Los atributos deben ser declarados al inicio de la clase antes de los métodos, constructores, inicializadores o clases internas.	A.65
16	Convenciones de nombrados de atributos.	Esta regla valida que los atributos de clase sean nombrados usando *camel case bajo* para atributos y *todas mayúsculas* para constantes (Porter, 2023).	A.66
17	Parámetro `final` en método abstracto.	Es inútil declarar un parámetro `final` en un método abstracto, dado que sus implementaciones podrían no respetarlo.	A.67
18	Ciclo `for` debería ser `while`.	Algunos ciclos `for` pueden ser simplificados mediante el uso de `while`, de tal forma hacerlos más concretos.	A.68

#	Regla	Descripción	Ejemplo
19	Convenciones de nombrado de parámetros formales.	Esta regla valida que los parámetros formales sean nombrados mediante el uso de *camel case bajo* (Porter, 2023).	A.69
20	Convenciones de nombrado de genéricos.	Las referencias de genéricos deben nombrarse con una única letra mayúscula.	A.70
21	Ramas de `catch` idénticas.	Más de 1 rama de `catch` que ejecuta el mismo código, incrementa la complejidad sin agregar nueva funcionalidad. Es preferible agrupar estas ramas en una sentencia de control *multi-catch*.	A.71
22	Nombrado de variables semántico.	Esta regla valida que el nombre de variables y métodos sea semánticamente coherentes (en inglés) a su tipo. Por lo tanto se validan las siguientes características: getters, setters, booleanos. atributos y variables.	A.72
23	Convención de nombrado para `LocalHome`.	Interfaz de un EJB `Localhome` debe contener el sufijo `LocalHome`.	A.73
24	Convención de nombrado para `LocalInterface`.	Interfaz de un EJB `LocalInterface` debe contener el sufijo `Local`.	A.74
25	Variable local puede ser `final`.	Una variable local que sólo ha sido asignada una vez, puede ser `final`.	A.75
26	Convención de nombrado para variable local.	Esta regla valida que las variables locales sean nombradas mediante el uso de *camel case bajo* (Porter, 2023).	A.76
27	Nombre de variable muy largo.	Variables, parámetros formales y/o atributos que tengan más de 17 caracteres pueden ser difíciles de leer y seguir.	A.77
28	Convención de nombrado para `MessageDrivenBean` y `SessionBean`.	Los `MessageDrivenBean` y `SessionBean` deben terminar con el sufijo `Bean`.	A.78
29	Parámetro formal puede ser `final`.	Un parámetro formal que sólo ha sido asignada una vez, puede ser `final`.	A.79
30	Convención para nombrado de métodos.	Esta regla valida que los métodos sean nombrados mediante el uso de *camel case bajo* (Porter, 2023).	A.80
31	Clase sin paquete especificado.	Detecta cuando las clases son declaradas sin un paquete asignado.	A.80
32	Clase sin paquete especificado.	Detecta cuando las clases son declaradas sin un paquete asignado.	A.81
33	Sólo un retorno.	Los métodos deben tener un único retorno y este debe ser la última línea del mismo.	A.82
34	Convención de nombrado de paquetes.	Los parques deben contener sólo caracteres alfa-numéricos en minúsculas.	A.83
35	Declaración prematura.	Valida que las variables no sean definidas prematuramente a su uso. Una variable es considerada como prematura si es creada antes de un bloque de código que contenga un retorno o alguna sentencia que evita que sea usada posteriormente.	A.84
36	Convención de nombrado para interfaces remotas.	Interfaces remotas de un EJB de sesión no deben tener sufijo.	A.85
37	Convención de nombrado para interfaces remotas.	Interfaces remotas de un EJB de sesión `RemoteHome` deben tener el sufijo `Home`.	A.86
38	Nombre de clase muy corto.	Nombres de clase con menos de 5 caracteres no son recomendados.	A.87
39	Nombre de método muy corto.	Nombres de métodos con menos de 3 caracteres no son recomendados.	A.88
40	Nombre de variable muy corto.	Nombres de variables con menos de 3 caracteres no son recomendados.	A.89
41	Demasiadas importaciones estáticas.	Si se sobre-usa el importe estático, puede hacer que el programa sea poco legible y no mantenible. Esta regla valida que no se tengan más de 4 importaciones estáticas.	A.90

#	Regla	Descripción	Ejemplo
42	Especificación de `value` en anotación innecesario.	Evite especificar `value` en una anotación cuando sólo hay un elemento.	A.91
43	*Cast* innecesario.	Esta regla identifica cuando se está haciendo un proceso de *Cast* cuando no es necesario, es decir, cuando el tipo retornado es el mismo de la variable al que se le asigna.	A.92
44	Constructor innecesario.	Cuando hay un sólo constructor y es idéntico al de defecto, y adicionalmente tiene el mismo modificador de la clase declarante por lo tanto no es necesario.	A.93
45	Nombre calificado completo innecesario.	No es necesario usar el nombre calificado completo de una clase, cuando ya fue importada.	A.94
46	Importación innecesaria.	Detecta cuando una importación es innecesaria, evalúa los siguientes casos: importaciones duplicadas, importaciones no usadas y miembros que ya son importados de forma implícita `java.lang`.	A.95
47	Variable innecesaria antes del retorno.	Detecta cuando se crea una variable que no es necesaria simplemente para retornarla inmediatamente después.	A.96
48	Modificador innecesario.	Los atributos en las interfaces son automáticamente `public static final` y los métodos son `public abstract`. Clases o anotaciones en una anotación de interfaz son automáticamente `public static`. Las enumeraciones son automáticamente `static`. Es innecesario declarar explícitamente los valores mencionados anteriormente.	A.97
49	Retorno innecesario.	Valida el uso innecesario de `return` en métodos de tipo `void`.	A.98
50	Punto y coma innecesario.	Reporta las sentencias vacías las cuales son generadas por `;` innecesarios.	A.99
51	Use el operador diamante.	Use el operador diamante `<>` para que java infiera el tipo automáticamente.	A.100
52	Paréntesis innecesario.	Paréntesis innecesarios deben ser removidos.	A.101
53	Calificador `this` innecesario.	Reporta el uso de `this` dentro de la misma clase.	A.102
54	Use el inicializador de arreglos corto.	Cuando se declara y/o inicializan atributos o variables de tipo arreglo, no es necesario el uso de `new` de forma explícita. Ejemplo: `int[] x = new int[] { 1, 2, 3 };` puede ser `int[] x = {1, 2, 3 };`.	A.103
55	Use guion bajo en literales de tipo numérico.	A partir de java 1.7 es posible el uso de guion bajo, para separar los dígitos en literales numéricos para mejorar la legibilidad.	A.104

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.3. Diseño Java

Reglas que le ayudan a descubrir problemas de diseño (PMD, 2022).

Tabla **3-5**.: Conjunto de reglas de diseño para Java

#	Regla	Descripción	Ejemplo
1	Clase abstracta sin ningún método.	Si una clase abstracta no contiene ningún método, puede estar diseñada como un simple contenedor de datos y no debería ser instanciada. Para este caso, es preferible usar un constructor privado o protegido para evitar su instanciación.	A.105

#	Regla	Descripción	Ejemplo
2	Evite capturar excepciones genéricas.	Evite capturar las siguientes excepciones: `NullPointerException`, `RuntimeException`, `Exception`, en bloques de tipo `try/catch`.	A.106
3	Evite el exceso de condicionales `if` anidados.	Evite el exceso de condicionales `if` anidados, debido a que son más complejos de entender y podrían generar errores difíciles de mantener.	A.107
4	Evite relanzar excepciones.	Bloques `catch` que simplemente relanzan excepciones, sólo agregan código y complejidad computacional..	A.108
5	Evite relanzar una nueva instancia de la misma excepción.	Bloques `catch` que simplemente crean una nueva instancia de la misma excepción, sólo agregan código y complejidad computacional.	A.109
6	Evite lanzar excepciones de tipo `NullPointerException`.	Lanzar excepciones de tipo `NullPointerException` generan confusión, toda vez que la mayoría de los desarrolladores pueden asumir que la máquina virtual de Java lanzó la excepción.	A.110
7	Evite lanzar excepciones genéricas.	Se recomienda evitar el lanzamiento de genéricas como: `Throwable`, `RuntimeException`, `Exception` y/o `Error`. Use una excepción y/o error específico.	A.111
8	Evite especificar excepciones no chequeadas en los métodos.	No es necesario agregar la cláusula `throws` en la firma de los métodos para excepciones no chequeadas como `RuntimeException`. Es preferible agregar ese nivel de detalle en el javadoc del método.	A.112
9	Clases con sólo constructores privados deberían ser de tipo `final`.	Las clases con constructores privados deben ser de tipo `final` a menos que el constructor privado sea invocado por una clase interna.	A.113
10	Complejidad cognitiva.	Los métodos que son altamente complejos son los más difíciles y costosos para leer y mantener. Si se incluye mucha lógica a partir de demasiadas decisiones en un mismo método, se genera muchos comportamientos difíciles de entender y modificar. La **complejidad cognitiva** es una medida que determina que tan difícil es un para los humanos entender y/o modificar un método (Ann et al., 2021).	A.114
11	Condicionales `if` agrupables.	En la mayoría de los casos dos `if` consecutivos pueden ser consolidados en uno sólo mediante el uso de un operador booleano como `and` o `or`.	A.115
12	Acoplamiento entre objetos.	Esta regla cuenta el número de atributos únicos, variables locales, y tipos de retorno en un objeto. Más de 20 puede indicar un alto grado de acoplamiento.	A.116
13	Complejidad ciclomática.	La complejidad de los métodos afecta directamente el costo de mantenimiento y legibilidad del código. Concentrar mucha lógica de decisiones en un único método genera complejidad al leer o modificar el código. La complejidad ciclomática evalúa la complejidad den un método contando la cantidad de puntos de decisión en un método más la entrada al método. Lo anterior incluye todas las sentencias de flujos de control como los son `if`, `while`, `for` y `case`.	A.117
14	Clase de datos.	Las clases de datos simplemente son contenedores, quienes revelan su estado sin ninguna funcionalidad compleja. La falta de funcionalidad indica que el comportamiento de la clase está definido en otra parte. Esta regla se encarga de evaluar que los operadores y la exposición de sus atributos deben hacerse en la misma clase y no en una clase externa.	A.118
15	No extienda `java.lang.Error`.	Las clases de tipo `Error` son excepciones del sistema, por lo tanto no se deben extender.	A.119
16	No usar excepciones como un flujo de control.	Usar excepciones como una forma de controlar el flujo de ejecución no es recomendad ya que puede esconder excepciones reales cuando se esté depurando el código. Prefiera el uso de condicionales, ciclos como flujo de control.	A.120
17	Excesiva cantidad de importaciones.	Un alto número de importaciones indica un alto grado de acoplamiento dentro de un objeto. Esta regla cuenta el número de importaciones únicas y reporta como violación si el número es mayor a 30.	A.121

#	Regla	Descripción	Ejemplo
18	Excesiva cantidad de parámetros.	Métodos con una cantidad numerosa de parámetros son difíciles de mantener, especialmente si la mayoría de ellos comparten el mismo tipo de dato. Esta situación denota la necesidad de agrupar los parámetros en un objeto común.	A.122
19	Conteo de elementos públicos excesivo.	Las clases con un número grande de métodos y atributos públicos (45 máximo), requieren un esfuerzo desproporcionado en términos de pruebas, toda vez que los efectos crecen rápidamente y el riesgo se incrementa. Dividir la clase en unidades más pequeñas incrementa la capacidad de pruebas y fiabilidad lo cual permite nuevas variaciones para ser desarrolladas más fácilmente.	A.123
20	Atributo `final` puede ser estático.	Si un atributo `final` es asignado en tiempo de compilación también podría ser estático, así se ahorra sobre carga en cada objeto en tiempo de ejecución.	A.124
21	"*Clase Divina*".	Una "*Clase Divina*" detecta fallas de diseño mediante el uso de métricas. Las "*Clases Divina*" son aquellas que hacen muchas cosas, son demasiado grandes y altamente complejas. Estas podrían dividirse y diseñarse orientadas a objetos. La regla usa la estrategia de detección descrita en el artículo "*Object oriented metrics in practice*" (Lanza and Marinescu, 2006).	No aplica.
22	Atributo inmutable.	Identificad cuando el valor de un atributo de clase privado nunca cambia. Esto ayuda en convertir la clase en una inmutable. Esta regla no fuerza a que una clase sea inmutable no obstante identifica miembros de la misma que podrían ser inmutables para aumentar su rendimiento.	A.125.
23	Bean de Java inválido.	Cada atributo no estático debe tener su "*getter*" y "*setter*". Si el atributo sólo es usado internamente y no es una propiedad de bean, entonces el atributo debe ser marcado como `transient`. La regla también verifica que el tipo del atributo es el mismo retornado por su método "*getter*" y modificado por el método "*setter*". La regla también valida, que existe un constructor por defecto o sin argumentos. Adicionalmente, la regla también verifica que el bean implemente `java.io.Serializable`.	A.126.
24	"*Principio de menor conocimiento*".	El "*principio de menor conocimiento*" es una regla que indica que una clase debe conocer lo menos posible a las clases a las que hace referencia. Así se reduce el acoplamiento entre las clases y/u objetos (Lieberherr and Holland, 2023).	A.127.
25	Inversión lógica.	Prefiera el uso del operador opuesto en condicionales `if` en vez de negar la expresión entera.	A.128.
26	Acoplamiento de paquete suelto.	Evite el alto acoplamiento entre clases de la misma jerarquía de paquetes.	A.129.
27	Estado estático mutable.	Atributos no privados estáticos deben ser constantes declarándolos `final`. Atributos no privados y no estáticos rompen el encapsulamiento y puede generar defectos difíciles de encontrar, toda vez que estos atributos pueden ser modificados en cualquier parte del programa. Los llamantes de la clase pueden trivialmente modificar los atributos no privados no estáticos. Esta regla detecta reporta este tipo de situaciones.	A.130.
28	Métrica de conteo de sentencias.	La métrica de conteo de sentencias (NCSS en inglés) determina el número de líneas de código en una clase. NCSS ignora comentarios, líneas en blanco y sólo cuenta sentencias reales.	A.131.
29	"*Complejidad NPath.*".	La "*Complejidad NPath*" de un método es un número de caminos de ejecución acíclicos que posee un método. Mientras que la complejidad ciclomática cuenta el número de puntos de decisión en un método. La "*Complejidad NPath*" de un método cuenta el número de caminos de inicio al fin del método. Esta regla tiene cómo límite un coeficiente de 200 así mejorar la legibilidad y mantenibilidad del código.	A.132.
30	Firma de método contiene `throws Exception`.	Un método y/o constructor no debe lanzar de forma explícita la excepción genérica `java.lang.Exception`, toda vez que no es claro qué tipo de excepciones específicas puede lanzar el método. Esto puede ser difícil de entender y/o documentar.	A.133.
31	Simplifique operadores ternarios.	Esta regla evalúa el uso innecesario de operadores ternarios, lo cual simplifica la legibilidad y mantenibilidad del código.	A.134.

#	Regla	Descripción	Ejemplo
32	Simplifique expresiones de tipo booleanas.	Evite comparaciones innecesarias en expresiones booleanas, ellas no tienen ningún propósito definido e impacta altamente la legibilidad.	A.135.
33	Simplifique retornos booleanos.	Evite condicionales `if-else` cuando retorne un booleano. el resultado del condicional puede ser retornado en su lugar.	A.136.
34	Simplifique condicionales en el uso de `instanceof`.	No es necesario validar si un objeto es `null` antes de un `instanceof`. La palabra clave `instanceof` retorna false cuando el argumento obtenido es `null`.	A.137.
35	Atributo singular.	Los atributos cuyo alcance se limita únicamente a un simple método no dependen del valor de la clase u objeto entero. Podrían ser convertidos a variables locales.	A.138.
36	Densidad de sentencia `switch`.	Un alto número de sentencias en un caso de un condicional `switch` indica que este está sobrecargado. Es mejor mover esta lógica a nuevos métodos y/o subclases basados en la variable del `switch`.	A.139.
37	Demasiados atributos.	Clases que contengan más de 15 atributos pueden ser muy pesadas y podrían ser rediseñadas para tener menos atributos, por ejemplo agruparlos en nuevos objetos. Por ejemplo atributos de ciudad, estado y país podrían ser agrupados en un objeto de Dirección.	A.140.
38	Demasiados métodos.	Clases que contengan más de 10 atributos pueden ser rediseñadas para reducir su complejidad y generar clases más granulares.	No aplica.
39	Innecesaria sobreescritura de método.	Sobrescribir métodos que únicamente llaman al método padre es innecesario.	A.141.
40	Use objetos para tener un API más limpio.	Cuando se escriben métodos públicos, se debe pensar en términos de API. Si el método es público, significa que otras clases lo van a usar, por lo tanto, es necesario ofrecer un API comprensible. Por ejemplo si se pasa una gran cantidad de información en varios parámetros de tipo `String`, podrían agruparse en un simple objeto, de tal forma ofrecer un API más limpio.	A.142.
41	Use clases utilitarias.	Clases que sólo contienen métodos estáticos, pueden ser convertidas en clases utilitarias. Esto no aplica para clases abstractas. Tener en cuenta que si se convierte en una clase utilitaria debe agregarse un constructor privado, para evitar la instanciación de la clase.	A.143.

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.4. Documentación Java

Reglas que están relacionadas con la documentación del código (PMD, 2022).

Tabla **3-6**.: Conjunto de reglas de documentación para Java

#	Regla	Descripción	Ejemplo
1	Contenido de comentario.	Valida que las palabras `idiot|jerk` no se encuentren en los comentarios de java.	A.144
2	Comentario requerido.	Valida que los métodos, atributos, y clases posean un comentario de javadoc.	A.145
3	Tamaño de comentario.	Esta regla valida que los comentarios no tengan más de 6 líneas y que cada línea no tenga más de 80 caracteres.	A.146
4	Constructor vacío sin comentario.	Valida que los constructores por defecto vacíos contengan un comentario, explicando el motivo.	A.147
5	Método vacío sin comentario.	Valida que los métodos vacíos contengan un comentario, explicando el motivo.	A.148

#	Regla	Descripción	Ejemplo

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.5. Código propenso a errores Java

Reglas para detectar construcciones rotas, extremadamente confusas o propensas a errores de tiempo de ejecución (PMD, 2022).

Tabla **3-7**.: Conjunto de reglas de detección de posibles errores para Java

#	Regla	Descripción	Ejemplo
1	Evite hacer asignaciones en operandos.	Evite asignaciones en operandos; esto puede hacer que el código sea más difícil de leer. En algún momento la asignación = podría ser confundido con el operador de igualdad ==.	A.149
2	Asignación a atributo estático no `final` .	Identifica un posible uso no seguro de un atributo estático.	A.150
3	Evite alteración de la accesibilidad.	Métodos como `getDeclaredConstructors()`, `getDeclaredMethods()`, y `getDeclaredFields()` retornan todos los constructores, métodos y atributos incluyendo los privados. A estos se les puede hacer accesible llamando el método `setAccessible(true)`. Lo anterior provee acceso a datos protegidos por el objeto, violando así, el principio de encapsulamiento. La regla valida alteraciones de accesibilidad mediante la llamada del método: `setAccessible(true)`.	A.151
4	Evite el uso de la palabra `assert` como un identificador.	Desde la versión de Java 1.4 el término `assert` es una palabra reservada, por lo tanto se prohíbe su uso como identificador.	A.152
5	Evite el uso de la palabra `assert` como un identificador.	Desde la versión de Java 1.4 el término `assert` es una palabra reservada, por lo tanto se prohíbe su uso como identificador.	A.152
6	Evite el uso de condicionales para continuar la ejecución de un ciclo.	Usar condicionales que contengan simplemente la sentencia `continue` para continuar con la ejecución, puede generar defectos difíciles de entender y detectar. Esta regla reporta este comportamiento para que se verifique si es un comportamiento deseado o es mejor cambiar de enfoque.	A.153
7	Evite el llamado explícito a `finalize()`.	Método `Object.finalize()` es llamado automáticamente por el recolector de basura de java, cuando este determina que no hay más referencias al objeto. No debe ser llamado directamente por la lógica de la aplicación.	A.154
8	Evite capturar la excepción `NullPointerException` .	La lógica de la aplicación no debería lanzar nunca `NullPointerExceptions` en circunstancias normales. Capturar `NullPointerExceptions` podría esconder errores generales de la aplicación, causando problemas mayores más adelante.	A.155
9	Evite capturar `Throwable`.	Capturar `Throwable` no es recomendado debido a que su alcance es muy amplio. Por ejemplo `Throwable` abarca errores de ejecución como `OutOfMemoryError` que debería manejarse de una forma especial.	A.156
10	Evite literales decimales en el constructor de `BigDecimal`.	Es incorrecto asumir que `new BigDecimal(0.1)` es exactamente igual a 0.1, cuando en realidad es igual a $0.1000000000000000055511151231 2$. Es preferible usar un literal de tipo String como `new BigDecimal("0.1")`, o en su defecto preferir el uso de números enteros.	A.157

#	Regla	Descripción	Ejemplo
11	Evite duplicar literales .	Es preferible crear una constante para el reuso de literales en vez de duplicarlos a lo largo de la aplicación.	A.158
12	Evite duplicar literales .	Es preferible crear una constante para el reuso de literales en vez de duplicarlos a lo largo de la aplicación.	A.159
13	Evite que el nombre de atributos/variables coincidan con el nombre de los métodos.	Es confuso tener un atributo/variable que tenga el mismo nombre de un método. Aunque es permitido nombrar una acción (método) e información (variable) con el mismo nombre no es claro.	A.160
14	Evite que el nombre de atributos/variables coincidan con el nombre de las clases/tipos.	Es confuso tener un atributo/variable que tenga el mismo nombre de una clase/tipo. Es preferible revisar con detalle el criterio para el nombrado de elementos de la aplicación.	A.161
15	Evite el uso de `instanceof` en la cláusula `catch`.	Cada excepción capturada debe manejarse en su propia cláusula `catch`.	A.162
16	Evite el uso de literales en condicionales `if`.	Evite el uso de literales fijos en condicionales. Es preferible declararlos como constantes, variables estáticas o miembros privados de la clase, que contengan nombres descriptivos con el fin de mejorar la legibilidad y mantenibilidad.	A.163
17	Evite perder información de las excepciones.	Las sentencias en el bloque de captura `catch` pueden ocultar la información de la excepción. Es preferible registrar la excepción en el `log` o en su defecto retornar lanzar la excepción, para que la información no se pierda.	A.164
18	Evite el uso de múltiples operadores unarios.	El uso de múltiples operadores unarios puede ser problemático y confuso. Asegúrese que es el comportamiento esperado o considere simplificar la expresión.	A.165
19	Evite el uso de valores octales.	Los literales enteros no deben iniciar con cero debido a que es interpretado por java como un valor Octal.	A.166
20	Incorrecta validación de `null` .	Se debe usar el operador && cuando se hace una validación de `null`, si se usa el operador `||`, es posible generar un `NullPointerException`.	A.167
21	Llame a `super` primero.	La sentencia `super` debe llamarse al inicio de un método sobrecargado.	A.168
22	Llame a `super` al final.	La sentencia `super` debe llamarse al final de un método sobrecargado.	A.169
23	Valide el resultado del método `skip()`.	El método `skip()` podría saltar menos bytes de los que se solicitó. Es necesario validar que el número necesario de bytes fue saltado.	A.170
24	Lanzamiento de `ClassCastException` cuando se usa `toArray`.	Cuando se deriva un arreglo a una clase específica de la interfaz `Collection`, se debe proveer un arreglo de la misma clase del parámetro del método `toArray()`, de lo contrario lanzará un `ClassCastException`.	A.171
25	Método `clone` debe ser público.	Por convención el manual de java especifica las clases que implementen la interfaz `Cloneable` deben sobreescribir `Object.clone()` con un método público.	A.172
26	Método `clone` debe implementar `Cloneable`.	El método `clone` sólo debe ser sobreescrito por clases que implementen la interfaz `Cloneable`.	A.173
27	Método `clone` debe retornar un tipo que coincida con la clase que los está implementando.	El método `clone` debe retornar un tipo que coincida con el nombre de la clase. Así, los llamantes del método, obtienen el objeto del tipo que esperan.	A.174
28	Cierre de recursos.	Asegúrese que recursos como `java.sql.Connection`, `java.sql.Statement`, `java.sql.ResulSet`, `java.util.Stream` y todos sus subtipos, etc. Se cierren dentro de un bloque `try/catch/finally`.	A.175
29	Compare igualdad de objetos usando `equals`.	Evite el uso del == para comparar igualdad entre objetos, es preferible usar el método `equals`.	A.176

#	Regla	Descripción	Ejemplo
30	Comparación con `NaN`.	Reporta comparaciones con `NaN` (Not a Number) para `double` y `float`. Comparar con `NaN` generar resultados inesperados, debido que `NaN` es considerado no igual a si mismo.	A.177
31	Constructor llama a un método sobre-escribible.	Llamar métodos sobre-escribibles durante la construcción tienen un riesgo de invocar métodos que no están completamente construidos y son complejos de depurar.	A.178
32	Test desprendido.	Esta regla reporta un método que parece ser un test debido a que es: público o package, no estático, no argumentos, no anotaciones, pero es un miembro de una o más clases de JUnit. Si es un método utilitario debe tener una visibilidad privada. Si es un test o debe ser ignorado debe contener la anotación `@Test` o `@Ignore`.	A.179
33	No llame al recolector de basura de forma explícita.	Llamadas a `System.gc()`, `Runtime.getRuntime().gc()`. y `System.runFinalization()` no son recomendadas, por lo tanto debe evitarse.	A.180
34	No extienda `java.lang.Throwable`.	Extienda `Exception` y/o `RuntimeException` en lugar de `Throwable`.	A.181
35	No llame de forma fija a `/sdcard`.	Para dispositivos Android, evite llamar `/sdcard`, use `Environment.getExternalStorageDirectory()` en su lugar.	A.182
36	No termine la máquina virtual de Java.	Las aplicaciones web no deben llamar a `System.exit()`, debido a que el contenedor es el encargado de parar la máquina virtual de java. La regla también valida llamadas equivalentes como: `Runtime.getRuntime().exit()` y `Runtime.getRuntime().halt()`.	A.183
37	No lance excepciones en un bloque `finally{}`.	Lanzar excepciones en un bloque `finally` es confuso debido a que su ejecución proviene de otras excepciones o defectos.	A.184
38	No importe paquetes provenientes de `sun.*`.	Evite importar paquetes provenientes de `sun.*`, debido a que no son portables y son susceptibles a cambios.	A.185
39	No use `float` para iterar en ciclos que usan índices.	No use flotantes para iterar en ciclos que usan índices. Si es obligatorio usar puntos flotantes es necesario validar la precisión, no obstante es preferible usar enteros para todos los casos.	A.186
40	Bloque de captura `catch` vacío.	Los bloques `catch vacíos` encuentra instancias donde una excepción es capturada pero no se ejecuta ninguna acción. En la mayoría de las circunstancias es un comportamiento no deseado.	A.187
41	Finalizador vacío.	Reporta cuando una clase implementa el método `finalize()` vacío, los cuales deben ser removidos. Se anunció que el uso de este método ha sido declarado obsoleto desde JDK 9.	A.188
42	Igualdad con `null`.	Comparar con `null` debe hacerse mediante el uso de == y no con el método `equals`.	A.189
43	Método `finalize()` no llama a `super.finalize()`.	Si el método `finalize()` es implementado, la última acción debe ser llamar a `super.finalize`.	A.190
44	Método `finalize()` sólo llama a `super.finalize()`.	Si el método `finalize()` es implementado, debe implementar más lógica más allá de llamar a `super.finalize`.	A.191
45	Método `finalize()` sobre-cargado.	No se debe sobre-cargar el método `finalize()`, genera confusiones y no es llamado por la máquina virtual de java.	A.192
46	Método `finalize()` debe ser protegido.	Cuando se sobre-escribe el método `finalize()`, este debe tener visibilidad `protected`.	A.193
47	Operaciones idempotentes.	Reporta operaciones idempotentes, es decir, que no tienen ningún efecto.	A.194
48	Falta de cambio implícito en sentencias `switch`.	Sentencias `switch` sin `break` o `return` para cada caso generan comportamientos inesperados. La regla valida que todos los casos tengan su respectiva sentencia de salida.	A.195

#	Regla	Descripción	Ejemplo
49	Evite instanciar usando `getClass()`.	Sentencias `switch` sin `break` o `return` para cada caso generan comportamientos inesperados. La regla valida que todos los casos tengan su respectiva sentencia de salida.	A.196
50	Formato de Log inválido.	La regla valida por mensajes de `sl4j` y `log4j` que contengan la misma cantidad de parámetros requeridos por el mensaje.	A.197
51	Incrementador de ciclo desordenado.	Los incrementadores de ciclos `for` desordenados donde se referencia más de un índice, son poco comunes y complejos de entender.	A.198
52	Ortografía de métodos JUnit.	En JUnit 3 el método `setUp` es usado para inicializar las entidades de datos usadas en los test unitarios. El método `tearDown` se encarga de limpiar la data para los test unitarios. Si estos dos métodos se escriben de forma incorrecta el motor de JUnit no ejecutará la lógica mencionada anteriormente.	A.199
53	Método `suite` para JUnit.	El método `suite` en un test unitario de JUnit debe ser público y estático.	A.200
54	Método con el mismo nombre de su clase.	Métodos no constructores no deben tener el mismo nombre de su clase.	A.201
55	Validación de `null` mal ubicada.	Las validaciones de `null` no deben estar mal ubicadas. Si la validación de `null` no es la primera comparación en el condicional, entonces es muy probable que se lance una `NullPointerException` cuando el valor del objeto comparado sea `null`.	A.202
56	`serialVersionUID` ausente.	Las clases que implementan la interfaz `java.io.Serializable` deben tener definido el atributo `serialVersionUID`.	A.203
57	Método estático ausente en clases no instanciables.	Si la clase tiene sus constructores privados para evitar su instanciación, entonces debe tener al menos un método estático, de lo contrario, la clase no podrá usarse.	A.204
58	Más de un `Logger`.	Normalmente sólo un `Logger` es definido por clase.	A.205
59	No `case` en caso de sentencia `switch`.	Aunque es permitido, no se deben tener etiquetas en casos de `switch` debido a que no es fácil distinguir, cual es un caso de `switch` o cuando es una etiqueta.	A.206
60	Clase no totalmente Serializable.	La regla reporta cuando una clase implementa la interfaz `java.io.Serializable`, sin embargo todos sus atributos no lo son. Si algún atributo de la clase no es serializable entonces debe marcarse como `transient`.	A.206
61	Asignación incorrecta de `null`.	Es considerada mala práctica asignar `null` a una variable más allá de su declaración, esto podría indicar una falla de diseño en el código o falta de entendimiento conceptual.	A.209
62	Sobreescriba ambos métodos: `equals` y `hashcode`.	Cuando una clase sobreescribe alguno de los métodos `equals` y/o `hashcode`, debe implementar ambos métodos o ninguno. Es una mala práctica implementar sólo uno de los dos y delegar la segunda función a su clase padre.	A.210
63	Implementación apropiada del método `clone()`.	Si se implementa el método `clone()` por ende se debe llamar `super.clone()`.	A.211
64	Implementación apropiada de `Logger`.	Los atributos de `Logger` deben ser privados y estáticos. Adicionalmente, deben llamar al `LoggerFactory` para obtener una instancia de `log` para la clase.	A.212
65	Retorne una colección vacía en vez de `null`.	Para cualquier método que retorne una colección como: arreglos, colecciones, mapas; se recomienda retornar una colección vacía en vez de `null`. Así, se evita la generación de la excepción `NullPointerException`	A.213
66	No retorne información en el bloque `finally`.	Retornar información en el bloque `finally` es considerada una mala práctica, toda vez que puede descartar o esconder excepciones.	A.214
67	`SimpleDateFormat` necesita del parámetro `Locale` .	Se recomienda para todas las instancias creadas de `SimpleDateFormat` que se agregue el parámetro `Locale`, así se genera el formato más adecuado para la zona horaria configurada en el sistema operativo.	A.215

#	Regla	Descripción	Ejemplo
68	*"Singleton"* de un sólo método.	Algunas clases pueden tener más de una versión sobrecargada del método `getInstance`, lo cual es considerado como una mala práctica, debido a que la instancia del objeto no está en cache, y puede generarse una nueva instancia en cada llamado, por lo tanto el patrón del Singleton no se cumple.	A.216
69	*"Singleton"* retorna una nueva instancia.	Una clase Singleton debe tener una sola instancia. No validar que hay una instancia creada y retornar una nueva es una violación del patrón Singleton.	A.217
70	EJB estático debe ser `final`.	Acorde con la especificación de JEE, un EJB estático debe ser constante declarándolo `final`.	A.218
71	Inicialización de `StringBuffer|StringBuilder`.	Caracteres `char` individuales en la inicialización de un `StringBuffer|StringBuilder` será convertido a un entero. Esto puede generar *buffers* más grandes de lo esperado.	A.219
72	Método `equals` sospechoso.	Reporta cuando el nombre del método es sospechosamente parecido a la implementación del método `Object.equals`, lo cual puede denotar una intención de sobre-escribirlo. No obstante, el método en cuestión no está sobre-escribiendo a `Object.equals` sino que lo está sobre-cargando. Esta regla reporta estos casos para evitar errores en el código que puedan generar defectos difíciles de identificar y resolver.	A.220
73	Método `hashcode` sospechoso.	Reporta cuando el nombre del método es sospechosamente parecido a la implementación del método `Object.hashcode`, lo cual puede denotar una intención de sobre-escribirlo. No obstante, el método en cuestión no está sobre-escribiendo a `Object.hashcode` sino que lo está sobre-cargando. Esta regla reporta estos casos para evitar errores en el código que puedan generar defectos difíciles de identificar y resolver.	A.221
74	Octal sospechoso.	Reporta cuando hay una posible definición de un octal en un literal de String. La especificación de Java denota que una secuencia de octal dentro un String, consisten en una barra invertida (slash invertido) seguido de dígitos octales.	A.222
75	Clase de Test sin ningún caso de prueba.	Clases con sufijos como: *"Test"*,*"Tests"* o *"TestCase"* son usualmente clases de prueba. Esta regla verifica en este tipo de clases que se tengan métodos con el patrón: `test<NombreDeTest>` para JUnit3 o con anotación `@Test`.	A.223
76	`if` incondicional.	Reporta casos en los condicionales `if` evalúan casos que son siempre falsos o verdaderos.	A.224
77	Aserción booleana innecesaria.	Una aserción que evalúe un literal booleano no sirve para ningún propósito. Es preferible reescribir la lógica de la prueba en vez de tener sentencias como `assert(true)` o `assert(false)`.	A.225
78	Transformación a mayúsculas o minúsculas innecesaria.	Es preferible usar `equalsIgnoreCase()` en vez usar `toUpper|toLower` para comparar literales de String, sin considerar sus mayúsculas y/o minúsculas.	A.226
79	Objetos temporales innecesarios para conversión.	No es necesario crear objetos temporales para convertir primitivos a String. Es preferible usar métodos con método estático para la conversión como: `Intreger.toString()`.	A.227
80	Validación de `null` innecesaria.	No es necesario validar `null` cuando luego el mismo objeto será enviado como parámetro al método `equals`.	A.228
81	Use la excepción correcta en `Loggger`.	Para asegurarse que una excepción quede bien registrada en el log debe pasarse tanto el mensaje como el objeto `Throwable`.	A.229
82	Use `equals` para comparar Strings.	Usar `==` o `!=` para comparar String puede conducir a resultados incorrectos. Prefiera el uso de `equals` para comparar Strings.	A.230
83	Operación idempotente en inmutables.	Una operación en un objeto inmutable como (String, BigDecimal o BigInteger) no cambiará el valor del objeto toda vez que el resultado de la operación es un nuevo objeto. Por lo tanto el retorno del resultado de la operación debe asignarse explícitamente al objeto inmutable para cambiar su valor.	A.231
84	Use `Locale` a la hora de convertir a mayúsculas o minúsculas.	Cuando se usa `String::toLowerCase()/toUpperCase()` se debe enviar el `Locale` para tener en cuenta la zona del sistema operativo.	A.232

#	Regla	Descripción	Ejemplo
85	Use `Locale` a la hora de convertir a mayúsculas o minúsculas.	Cuando se usa `String::toLowerCase()/toUpperCase()` se debe enviar el `Locale` para tener en cuenta la zona del sistema operativo.	A.232
86	Use el `ClassLoader` apropiado.	Para todos los casos debe usarse `Thread.currentThread().getContextClassLoader()`, el método `getClassLoader()` no puede funcionar como se espera para todas las máquinas virtuales de Java.	A.233

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.6. Multi-hilo Java

Reglas que marcan problemas cuando se trata de varios subprocesos de ejecución (PMD, 2022).

Tabla **3-8**.: Conjunto de reglas de detección de multi-hilo para Java

#	Regla	Descripción	Ejemplo
1	Evite usar `synchronized` a nivel de método.	Usar `synchronized` puede generar problemas de rendimiento especialmente cuando se agregue nuevo código a futuro. Se recomienda usar bloques sincronizados en su lugar.	A.234
2	Evite usar `java.lang.ThreadGroup`.	Evite el uso de `java.lang.ThreadGroup`; si bien su uso está destinado para manejar múltiples hilos, contiene métodos que no son hilo seguro.	A.235
3	Evite el uso de `volatile`.	El uso de la palabra clave `volatile` es usado para aumentar el rendimiento de una aplicación Java, sin embargo, requiere un alto nivel de pericia con respecto al modelo de Memoria de Java. Teniendo en cuenta lo anterior, se recomienda que la palabra clave `volatile` no sea usado en pro de la mantenibilidad y portabilidad.	A.236
4	Evite el uso de `Threads`.	La especificación de JEE explícitamente prohíbe el uso de `Threads`. Los `Threads` son recursos que deben ser manejados y monitoreados por JEE. Si la aplicación crea su propio conjunto de hilos, estos hilos no son administrados y podrían conducir a agotamiento de recursos de la máquina virtual de Java.	A.237
5	Evite el uso de `Thread.run()`.	El llamado explícito a `Thread.run()` se ejecutará en el hilo de ejecución del llamante sin ningún control específico. Se recomienda llamar `Thread.start()` en su lugar.	A.238
6	Patrón de doble bloqueo.	Se recomienda usar `volatile` para bloques sincronizados para optimizar la instanciación y acceso de objetos, cumpliendo el patrón de doble bloqueo.	A.239
7	Singleton inseguro para multi-hilo.	Singleton para multi-hilos no deben generarse mediante métodos estáticos si es posible instanciar el objeto directamente. Los Singleton estáticos no son necesarios porque una sola instancia del objeto ya existe.	A.240
8	Formato de fecha no sincronizado.	Instancias y sub-clases de `java.text.Format` usualmente no son sincronizadas. Se recomienda crear Formatos de fecha y tiempo en bloques sincronizados para soportar arquitecturas multi-hilo.	A.241
9	Use `ConcurrentHashMap` para arquitecturas multi-hilo.	En la versión de Java 5 se introdujo el diseño de la interfaz `Map` para soportar arquitecturas `multi-hilo`, se recomienda usar `ConcurrentHashMap` para este propósito.	A.242

#	Regla	Descripción	Ejemplo
10	Use `notifyAll` en vez de `notify`.	El método `Thread.notify()` despierta un hilo que esté monitoreando un objeto. No obstante, se recomienda el uso `Thread.notifyAll()` para notificar al tiempo a todos los hilos que estén en estado de monitoreo.	A.243

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.7. Rendimiento Java

Reglas que marcan el código subóptimo (PMD, 2022).

Tabla **3-9**.: Conjunto de reglas de rendimiento para Java

#	Regla	Descripción	Ejemplo
1	No sume un `String` vacío `""` para conversión de literales a `String`.	La conversión de literales a `String` mediante el uso de la adición de `""` es ineficiente. Se recomienda usar el método `toString()` del objeto o primitivo se desea convertir.	A.244
2	No concatene caracteres simples con literales de `String` para `StringBuffer|StringBuilder`.	Evite concatenar caracteres simples como `String` para `StringBuffer|StringBuilder`.	A.245
3	Evite ciclos manuales sobre arreglos para generar copias.	En vez de usar ciclos manuales para copiar un arreglo a otro, prefiera el uso de `Arrays.copyOf` y `System.arraycopy` los cuales son más eficientes.	A.246
4	Evite la creación de `Calendar` para obtener fechas y tiempo.	El objeto `Calendar` es muy pesado y costoso para crear. Se recomienda usar `Date()` y `java.time.LocalDateTime.now()` y `ZoneDateTime.now()` para Java 8.	A.247
5	Evite el uso de `FileStream`.	Las clases `FileInputStream` y `FileOutputStream` contienen métodos `finalizer` que generan pausas en el recolector de basura de java. Se recomienda el uso de: `InputStream`, `OutputStream`, `BufferedReader`, `BufferedWriter` en su lugar.	A.248
6	Evite instanciar objetos en ciclos.	Se prefiere revisar la lógica de la aplicación para determinar si se puede crear la instancia fuera del ciclo y reusarla dentro del mismo.	A.249
7	No cree instancias de constantes de `BigInteger`.	No se deben crear instancias para constantes de `BigInteger` como: `BigInteger.ZERO`, `BigInteger.ONE`, `BigInteger.TEN`.	A.250
8	Anexos (`append`) consecutivos deben reusarse en la misma sentencia.	Llamados consecutivos al método `append` de `StringBuffer|StringBuilder` deben ser encadenados en la misma sentencia. Así se mejora el rendimiento generando bytecode más pequeño que reduce la sobrecarga en ejecución y compilación.	A.251
9	Anexos (`append`) de literales consecutivos deben hacerse en la misma cadena.	Agregar letra por letra en literales consecutivos genera problemas de rendimiento, se recomienda anexar la palabra entera en una sola cadena de texto.	A.252

#	Regla	Descripción	Ejemplo
10	Chequeo ineficiente de un String vacío.	Usar métodos como String.trim().length == 0 o String.trim.isEmpty() es un modo ineficiente de validar si un String contiene sólo caracteres en blanco. Se recomienda que se cree un ciclo que valide cada caracter del String usando el método Character.isWhitespace()	A.253
11	Creación de StringBuffer|StringBuilder ineficiente.	Evite concatenar literales y no literales de Strings dentro del método append() en un StringBuffer|StringBuilder. Encadene el método append() por cada elemento que se desea anexar.	A.254
12	Inicialización de StringBuffer|StringBuilder ineficiente.	Definir un tamaño inicial de buffer muy pequeño para una cadena de caracteres muy grande, genera que la máquina virtual de Java cambie el tamaño varias veces durante el tiempo de ejecución, lo cual es ineficiente. Esta regla valida que el tamaño inicial del buffer corresponda al tamaño de la inicialización del StringBuffer |StringBuilder.	A.255
13	Llamada a toCallArray optimizable.	Cuando se llama al método toCallArray se debe especificar un arreglo de tamaño cero (0). Esto permite que la máquina virtual de Java optimice la asignación de memoria, y pueda copiar tantos elementos como le sea posible.	A.256
14	Inicializador de atributo redundante.	Java inicializa los atributos con los siguientes valores por defecto: **boolean** = false; **byte, char, int, short, long** = 0; **float** = .0f, **double** = 0d, **Object** = null. Por lo tanto, la inicialización explícita con los mismos valores es redundante y no agrega ningún valor.	A.257
15	Instanciación de String.	Evite la instanciación del objeto String, es innecesario debido a que son objetos inmutables y pueden compartirse con toda seguridad.	A.258
16	No use String. toString.	No es necesario llamar al método toString de una variable/atributo de tipo String, es una conversión innecesaria.	A.259
17	Muy pocos casos para una sentencia switch.	El condicional switch se usa para soportar condicionales complejos con varios casos, generados a partir de una variable y/o condición. Usar switch para soportar menos de 2 casos es innecesario, prefiera el uso de if/else.	A.260
18	Use ArrayList en vez de Vector.	Si no es necesario soportar arquitecturas multi-hilo, prefiera el uso de ArrayList en vez de Vector.	A.261
19	Use Arrays.asList().	El método asList() de la clase java.util.Arrays es usado para convertir un arreglo a una lista. Es más rápido usar este método, a crear un ciclo for que copie manualmente los elementos de un arreglo a una lista.	A.262
20	Use indexOf(char).	Prefiera el uso de indexOf(char) sobre indexOf(String), para buscar caracteres únicos dentro de una cadena más larga para mejorar el rendimiento.	A.263
21	Use IOStreams con FileItem de apache commons.	Las clases FileItem.get() y FileItem.getString() tienden a agotar la memoria de la máquina virtual de java. Prefiera el uso de FileItem.getInputStream() y BufferedReader , BufferedWriter.	A.264
22	String.valueOf innecesario.	No es necesario usar el método String.valueOf para anexar un String. Prefiera el uso de toString que tiene un mejor rendimiento.	A.265
23	Use StringBuffer|StringBuilder para concatenar String.	Las clases StringBuffer|StringBuilder tienen un mejor rendimiento al concatenar String que el uso de la agregación + .	A.266
24	Use StringBuffer |StringBuilder. length.	Use el método length de StringBuffer|StringBuilder para determinar si el buffer está vacío. Convertir a String y comparar contra la cadena vacía "" es ineficiente.	A.267

#	Regla	Descripción	Ejemplo

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.2.8. Seguridad Java

Reglas que marcan posibles fallas de seguridad (PMD, 2022).

Tabla **3-10**.: Conjunto de reglas de seguridad para Java

#	Regla	Descripción	Ejemplo
1	Llave criptográfica fija.	No use valores fijos para operaciones criptográficas. Guarde las llaves fuera del código fuente.	A.268
2	Vector de inicialización inseguro.	No use vectores de inicialización fijos para operaciones criptográficas. Use un vector de inicialización aleatorio.	A.269

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.3. Análisis de código Javascript

SAP Commerce soporta Javascript para el desarrollo de la capa de presentación de la tienda en línea. Por lo tanto en esta sección se describen reglas de buenas prácticas 3.3.1, estilo de código 3.3.2 y código propenso a errores 3.3.3 establecidos por la comunidad (PMD, 2022).

3.3.1. Buenas prácticas JavaScript

Reglas que refuerzan buenas prácticas generalmente aceptadas para JavaScript (PMD, 2022).

Tabla **3-11**.: Conjunto de reglas de buenas prácticas de JavaScript

#	Regla	Descripción	Ejemplo
1	Evite el uso de la sentencia `with`.	No se recomienda el uso de la sentencia `with`, ya que puede ser la fuente de errores confusos: no puede determinar a qué se refiere el identificador observando su entorno sintáctico.	A.270
2	Retorno consistente.	JavaScript proporciona distintos tipos de retorno en funciones y, por lo tanto, no existe una regla sólida en cuanto a su uso. Sin embargo, cuando una función usa retornos, todos deben tener un valor, o todos sin valor. Es probable que el uso de retorno mixto sea un error o, en el mejor de los casos, un estilo de codificación deficiente.	A.271

#	Regla	Descripción	Ejemplo
3	Variables Globales.	Esta regla ayuda a evitar el uso accidental de variables globales al simplemente omitir la declaración `var`. Las variables globales pueden provocar efectos secundarios que son difíciles de depurar.	A.272
4	Alcance de variables en ciclos `forin`.	Un ciclo `forin` en el que el nombre de la variable no se limita explícitamente al alcance con la palabra clave `var` puede hacer referencia a una variable en un ámbito envolvente fuera del alcance del `for in`. Esto sobrescribirá el valor existente de la variable en el ámbito externo cuando se evalúe el cuerpo del `for in`. Cuando el ciclo `for in` haya terminado, la variable contendrá el último valor utilizado en el `for in`, y el valor original anterior al bucle `for in` desaparecerá. Dado que el nombre de la variable `for in` probablemente tenga la intención de ser un nombre temporal, es mejor incluir explícitamente el nombre de la variable en el ámbito más cercano con `var`.	A.273
5	Defina la base a la hora de utilizar `parseInt`.	Esta regla comprueba los usos de `parseInt`. Si bien el segundo parámetro es opcional y generalmente tiene un valor predeterminado de 10 (la base/base es 10 para un número decimal), las diferentes implementaciones pueden comportarse de manera diferente. También mejora la legibilidad, si se da la base.	A.274

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.3.2. Estilo de código de JavaScript

Reglas que imponen un estilo de codificación específico que sea legible (PMD, 2022).

Tabla **3-12**.: Conjunto de reglas de estilo de codificación para JavaScript

#	Regla	Descripción	Ejemplo
1	Evite hacer asignaciones en operandos.	Evite asignaciones en operandos; esto puede hacer que el código sea más difícil de leer. En algún momento la asignación = podría ser confundido con el operador de igualdad ==.	A.275
2	Ciclos `for` deben usar llaves.	Evite usar ciclos `for` sin llaves.	A.276
3	Condicionales `if`|`else` deben usar llaves.	Evite usar condicionales `if`|`else` sin llaves.	A.277
4	Condicionales `if` deben usar llaves.	Evite usar condicionales `if` sin llaves.	A.278
5	Evite el uso de `else` -> `return`.	Si el condicional `if` ya contiene una sentencia de retorno, no es necesario agregar el bloque `else`. En su lugar puede poner la lógica restante fuera del `if`.	A.279
6	Evite el uso de bloques innecesarios.	Reporta el uso de un bloque innecesario. En otros lenguajes los bloques anónimos son usados para la introducción de un nuevo alcance de variables, lo cual no pasa en JavaScript y puede ser engañoso. Considere remover bloques que son innecesarios.	A.280
7	Evite el uso de paréntesis innecesarios.	JavaScript permite el uso de paréntesis innecesarios siempre y cuando se abran y cierren en el orden correcto. Esta regla reporta el uso de paréntesis innecesarios.	A.281
8	Reporte de código inalcanzable.	Las palabras clave `return`|`throw`|`break`|`continue` deben encontrarse al final del alcance. En caso de no estarlo las sentencias subsiguientes no se ejecutarán, lo cual genera defectos o un pobre estilo de programación.	A.282

#	Regla	Descripción	Ejemplo
9	Ciclos `while` deben usar llaves.	Evite usar ciclos `while` sin llaves.	A.283

Fuente: Elaboración propia y adaptación de (PMD, 2022)

3.3.3. Código propenso a errores de JavaScript

Reglas para detectar código roto, extremadamente confuso o propenso a errores en tiempo de ejecución.

Tabla **3-13**.: Conjunto de reglas de código propenso a errores para JavaScript

#	Regla	Descripción	Ejemplo
1	Evite la coma final en arreglos.	Esta regla ayuda a mejorar la portabilidad de código entre navegadores, toda vez, que los mismos tratan de forma diferente la coma al final de los arreglos y/o matrices.	A.284
2	Prefiera el uso de === sobre == a la hora de comparar igualdad.	El uso de == en comparación de igualdad puede generar resultados inesperados, ya que las variables se convierten automáticamente para que sean del mismo tipo. El operador === evita la conversión y compara el valor y el tipo de las variables comparadas	A.285
3	Evite proporcionar un literal numérico con mucha precisión.	Los literales numéricos en decimales o flotantes de alta precisión podrían tener un valor diferente en tiempo de ejecución, lo cual genera cálculos numéricos erróneos. Prefiera el uso de enteros o decimales de baja precisión cuando sea posible.	A.286

Fuente: Elaboración propia y adaptación de (PMD, 2022)

4. Implementación estrategia de análisis de código en SAP Commerce

Este capítulo contiene los detalles técnicos de la herramienta de software que se desarrolló con el fin de aplicar la estrategia de análisis automático de código fuente planteada en el Capítulo 3. A continuación, en la Sección 4.1 se presentan las características de la herramienta de análisis de código fuente de SAP Commerce. Estos detalles se relacionan directamente con la estrategia propuesta. Posteriormente, en la Sección 4.2 se detalla el flujo de ejecución de la herramienta, el cual comprende las fases de instalación (Sección 4.2.1), inicialización (Sección 4.2.2) y análisis (Sección 4.2.3).

4.1. Herramienta para el análisis de código fuente de SAP Commerce

En esta fase del proyecto se trabajó en la implementación de una herramienta de software que automatiza el análisis de código fuente para SAP Commerce. Esta herramienta se construyó teniendo en cuenta las siguientes singularidades:

- **Código abierto:** La herramienta se desarrolló mediante el uso de la licencia de Apache 2.0, la cual **permite:** el uso comercial, modificación, distribución, uso en patentes y uso privado. Adicionalmente, esta licencia **prohíbe:** uso de marca registrada, exime de responsabilidades al creador del código y, por último, no genera ningún tipo de garantía y/o vínculo comercial con los usuarios.

  El código fuente de la herramienta puede ser encontrado en el siguiente repositorio público

de github: `https://github.com/roger8849/sap_commerce_code_analyzer`. Por último, las librerías usadas por la herramienta gozan del mismo privilegio de ser de código abierto y/o de uso libre como se detalla en la Tabla **4-1**.

- **Flexible:** La estrategia plantea varias fases de análisis como se indica en el Capítulo 3. Esta herramienta fue diseñada para que la ejecución de cada una de ellas sea opcional.
- **Extensible:** Cada fase del análisis fue creada de forma modular. Por lo tanto, si se desean agregar más funcionalidades, existe la posibilidad de añadir otro módulo con código personalizado.

En la Tabla **4-1** se relacionan las herramientas usadas para la implementación donde se especifican las siguientes cualidades:

- **Herramienta:** Nombre de la herramienta y/o librería que se usó.
- **Versión:** Versión utilizada.
- **Licencia:** Tipo de licencia de la herramienta.
- **Descripción:** Explicación breve de la funcionalidad de la herramienta y su uso dentro del proyecto.

Tabla **4-1**.: Herramientas usadas en la implementación de la herramienta de análisis de código

Herramienta	Versión	Licencia	Descripción
Java OpenJDK versión 11.0	11	GNU General Public License Version 2 (OracleCorp, 2023)	Implementación de código abierto del lenguaje de programación Java. Este lenguaje de propósito general es el utilizado para la implementación. Las demás herramientas en esta tabla son compatibles con Java.
Gradle	8.0.1	Apache License Version 2 (GradleInc, 2023a)	Gradle es una herramienta de automatización de código abierto flexible que permite construir y compilar cualquier tipo de software basado en la máquina virtual de Java. Es usado en la implementación como herramienta de construcción, ejecución y modularización del proyecto.
PMD	8.0.1	Apache license version 2 (PMD, 2023)	PMD es un analizador de código estático el cual encuentra errores comunes de programación como variables sin usar, objetos creados innecesariamente, entre otros. Es principalmente usado para analizar los lenguajes Java y Apex, no obstante soporta otros 14 lenguajes de programación. Esta herramienta es usada como analizador base de código fuente, el cual fue extendido teniendo en cuenta la estrategia planteada en el Capítulo 3

Herramienta	Versión	Licencia	Descripción
Apache commons	3.12	Apache license version 2 (Apache, 2023)	Es un proyecto de la organización Apache que se enfoca en proveer componentes reusables para Java. Este conjunto de librerías se usa para manejo de cadenas de texto, lectura y escritura de archivos, manipulación de estructuras de datos entre otras funcionalidades.
SLF4J (Simple Logging Facade for Java) Simple log 4j	2.0.7	MIT License (SLF4J, 2023)	SLF4J sirve como una abstracción de varios sistemas de registro como: java.util.logging, logback o log4j. Es usado en la implementación como herramienta de registro de errores en la consola de ejecución.

Fuente: Elaboración propia.

La herramienta comprende la construcción de múltiples proyectos de Gradle (GradleInc, 2023c) tal y como se puede observar en el repositorio público de GitHub `https://github.com/roger8849/sap_commerce_code_analyzer`. Lo anterior implica que se creó un proyecto padre el cual se denominó como `sap_commerce_analyzer`. Los módulos de Gradle pueden ser de dos tipos: **Aplicación,** el módulo contiene un elemento ejecutable, es decir, una clase que implementa el método `main` de Java; **Librería,** el módulo contiene clases que pueden ser utilizadas o llamadas por otras clases, no obstante, no tienen un ejecutable que implemente el método `main` de Java.

El módulo `sap_commerce_analyzer` contiene el sub-módulo de aplicación `app` y los sub-módulos de librería `buildSrc, pmd-common, utilities, deprecated-extension-analysis, commerce-incompatibilities-analysis, third-party-libraries-analysis database-keyword-commerce-analysis, java-commerce-analysis, javascript-commerce-analysis` como se puede observar en el análisis de dependencias de Gradle de la Figura **4-1**. Estos sub-proyectos son explicados en detalle en las secciones posteriores.

4.1.1. Modulo genérico *app*

En la Figura **4-1** se observa el diagrama de dependencias de Gradle para el módulo app. Los elementos con el ícono de Gradle (elefante) son sub-proyectos de `sap_commerce_analyzer`. Por otra parte, los nodos que contienen los íconos de barras verticales son librerías de Java externas usadas por el módulo.

El módulo `app` es el único de tipo *aplicación*, el cual cumple con la función de iniciar la ejecución del

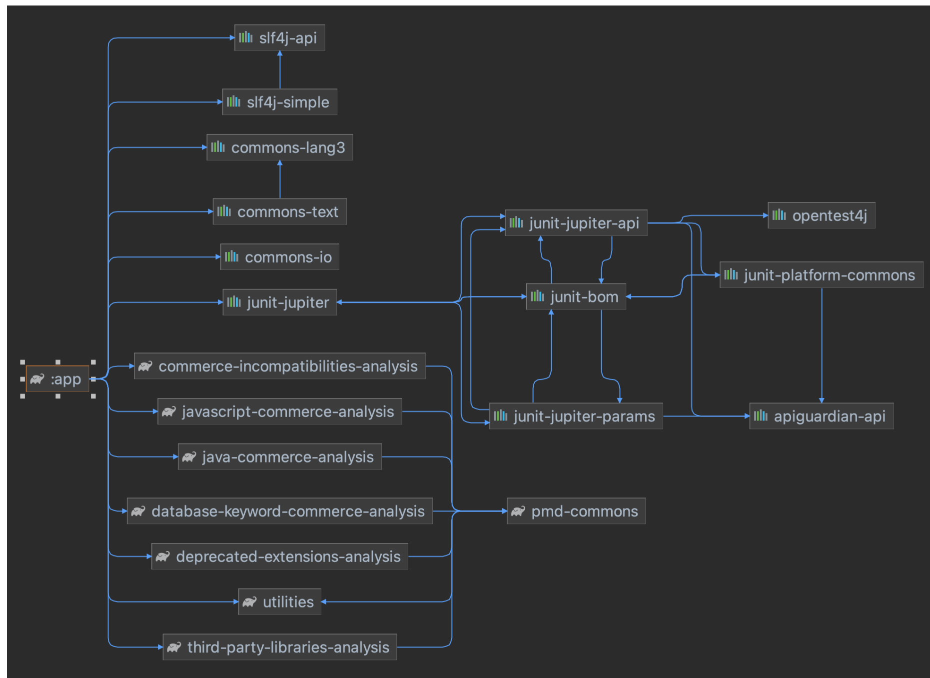

Figura **4-1**.: Diagrama de dependencias de Gradle del módulo app.

analizador de código fuente. Únicamente contiene una única clase llamada `App` (Figura **4-2**), la cual cumple con las siguientes funciones:

- Implementar el método de Java `main` el cual inicia el hilo principal de ejecución.
- Llamar a la clase `InitUtils` del módulo `utilities` (sub-sección 4.1.2) para obtener los parámetros de inicialización del análisis.
- Iniciar el análisis una vez los parámetros de inicialización han sido obtenidos.

Adicionalmente, como se observa en la Figura **4-1**, este módulo depende de las librerías mencionadas en la Tabla **4-1**, y depende de todos los demás sub-módulos que implementan las fases de la estrategia planteada en el Capítulo 3. Para mayores detalles de la implementación se recomienda consultar el código fuente del repositorio público de Github: `https://github.com/roger8849/sap_commerce_code_analyzer`.

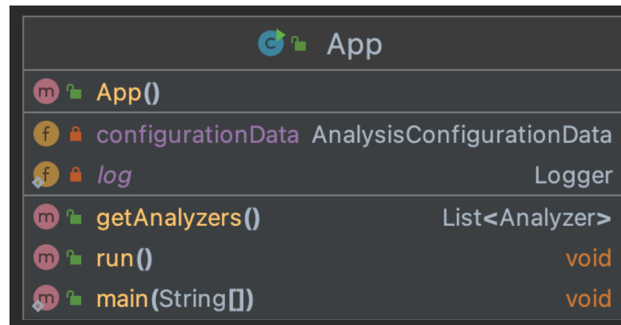

Figura **4-2**.: Diagrama de clases del módulo *app*.

4.1.2. Modulo genérico *utilities*

El módulo `utilities` es de tipo *librería*. Cumple con la función de definir clases estáticas que proveen métodos y objetos utilitarios que son usados por los demás módulos que ejecutan el análisis. Debido a que este sub-proyecto es utilitario, no depende de ningún otro módulo como se observa en la Figura **4-3**, sin embargo, utiliza herramientas y librerías que se especificaron en la Tabla **4-1**.

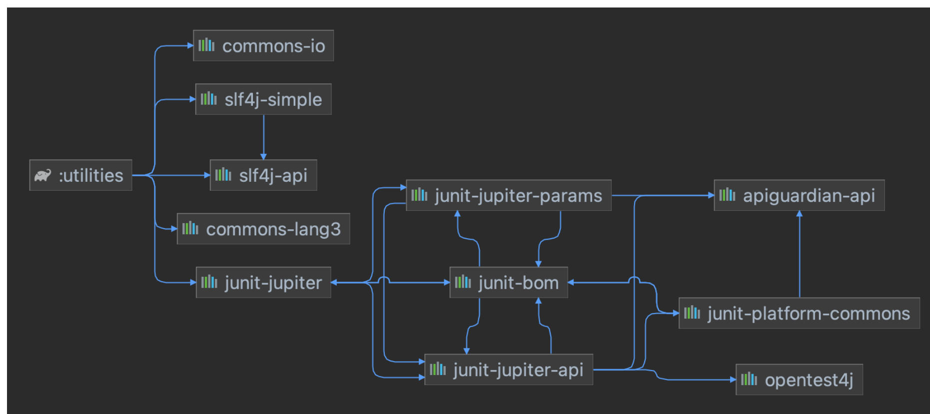

Figura **4-3**.: Diagrama de dependencias de Gradle del módulo *utilities*.

Este módulo contiene las clases que se observan en la Figura **4-4**, las cuales cumplen con las siguientes funciones:

- `AnalysisConfigurationData`: Objeto que almacena los datos de configuración que se obtuvieron durante la fase de inicialización (Sección 4.2.2). Este objeto contiene la siguiente información: ruta en el sistema de archivos donde se encuentra el código de SAP Commerce a analizar, ruta en el sistema de archivos donde se escribirá el resultado del análisis, variables que deter-

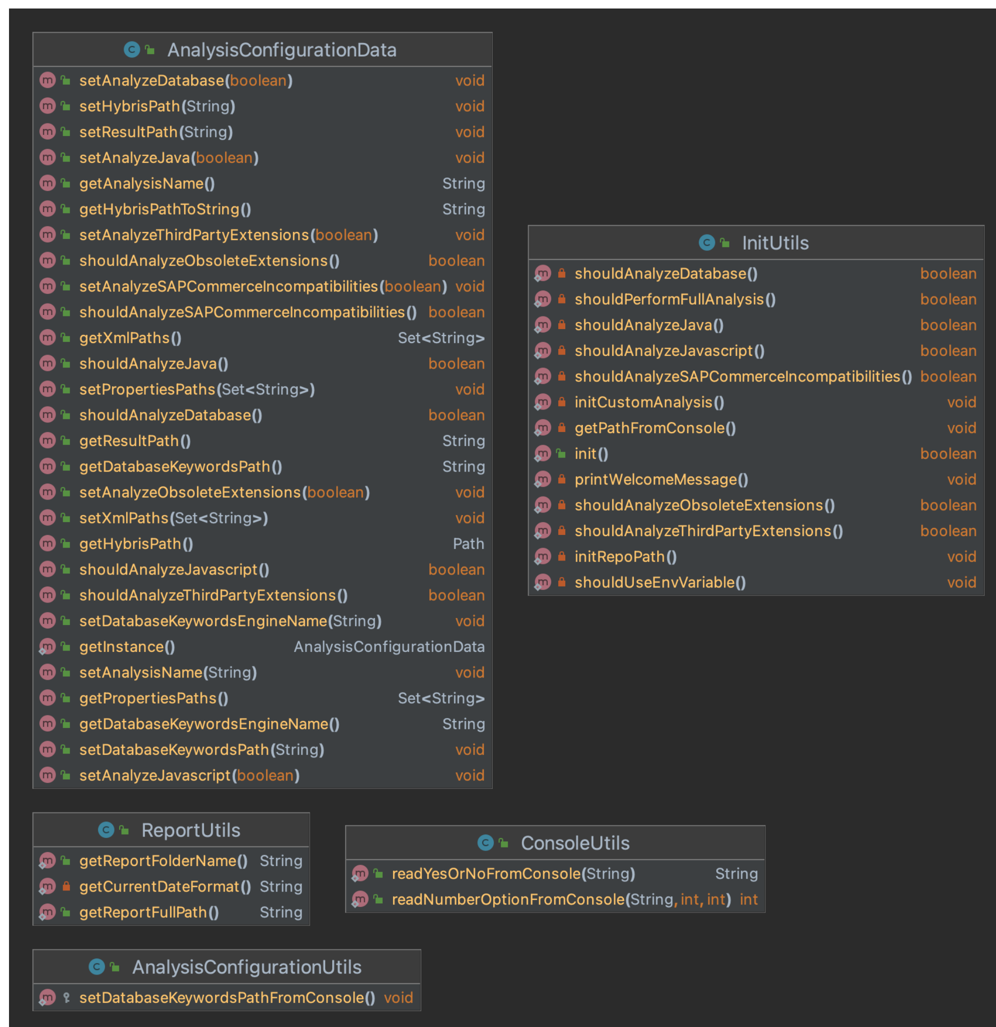

Figura **4-4**.: Diagrama de clases del módulo *utilities*.

minan cuáles fases del análisis deben ejecutarse, entre otras variables de configuración usadas por el programa.

- `InitUtils`: Esta clase es llamada por el módulo app (Sección 4.1.1) y obtiene las variables de inicialización del programa que se almacenarán en el objeto `AnalysisConfigurationData`. El objetivo principal es el de obtener toda la información necesaria para analizar el código fuente de SAP Commerce. Esta clase se encarga de invocar y leer la siguiente información que debe ingresar el usuario:
    - *Imprimir en consola mensajes de bienvenida.*
    - *Invocar lectura de parámetros desde la consola*. Invoca la clase `ConsoleUtils` con el fin de leer parámetros por consola.
    - *Ruta donde se encuentra el código fuente de SAP Commerce a analizar*. Esta ruta se lee desde la variable de ambiente de sistema operativo `HYBRIS_HOME`[1] si esta no contiene ningún valor, se pide que se ingrese por consola la ruta absoluta del sistema de archivos.
    - *Ejecutar análisis completo*. La clase se encarga de preguntarle al usuario a través de la consola, si desea realizar un análisis completo de todas las fases. En caso de seleccionar que no, el usuario debe indicar a través de la consola cuáles son las fases que desea analizar.
- `ConsoleUtils:` Contiene dos métodos utilitarios para leer parámetros por consola. El primer método se encarga de leer y validar una opción numérica entre uno y un límite superior. El segundo se encarga de leer cuando el usuario debe responder `yes|no`, con el fin de determinar cuáles análisis deben analizarse (Sección 4.2.2).
- `ReportUtils:` Contiene métodos utilitarios que se encargan de crear la carpeta que contendrá los archivos con el resultado del análisis (Sección 4.2.2).
- `AnálisisConfigurationUtils:` Contiene métodos utilitarios para leer archivos de propiedades que existentes dentro del proyecto, por ejemplo, las palabras clave de los motores de base de datos a encontrar por la herramienta.

[1] La variable de ambiente HYBRIS_HOME es comúnmente utilizada por los desarrolladores de SAP Commerce. Frecuentemente se usa para la ejecución de ciertos scripts de la herramienta.

4.1.3. Modulo genérico *pmd-commons*

El módulo `pmd-commons` es de tipo *librería*. Cumple con la función de definir objetos y librerías reutilizables para todos los demás módulos. En la Figura **4-5** se observa que el módulo `pmd-commons` depende del módulo genérico `utilities` (Sección 4.1.2). Esta dependencia permite que por transitividad los módulos que dependan de `pmd-commons`, puedan utilizar las clases presentes en `utilities`. Adicionalmente, este módulo tiene la dependencia a la librería `pmd-core` necesaria para usar la herramienta de análisis de código fuente PMD (PMD, 2022).

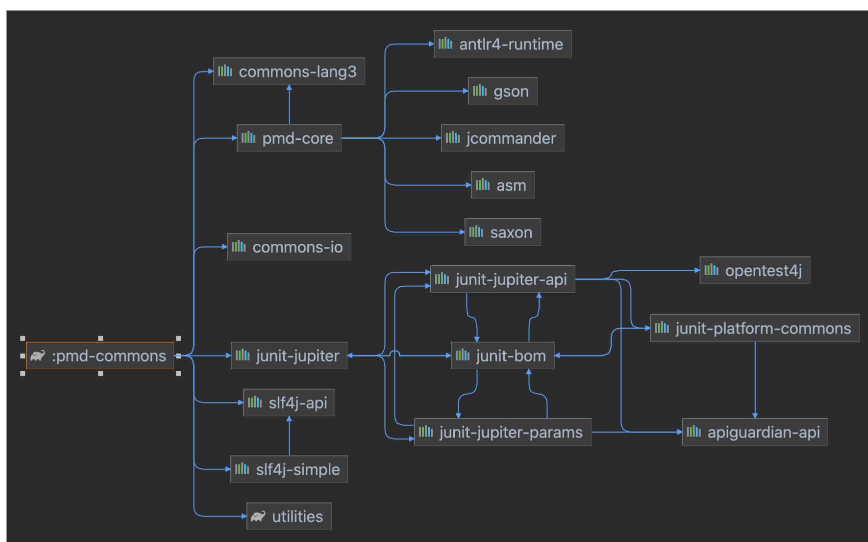

Figura **4-5**.: Diagrama de dependencias de Gradle del módulo *pmd-commons*.

`Pmd-commons` declara la interfaz `Analyzer` (Figura **4-6**) que especifica dos métodos: `analyze` y también `analyzerConfiguration`. Estos métodos son implementados para ejecutar cada análisis con el método `analyze` con la configuración `getObjectConfiguration`.

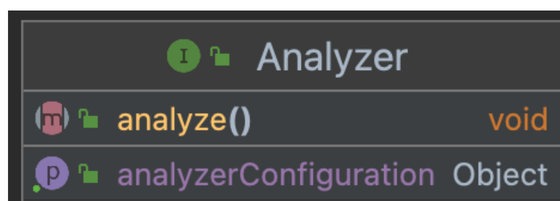

Figura **4-6**.: Diagrama de clases del módulo *pmd-commons*.

4.1.4. Módulo genérico *buildSrc*

El módulo `buildSrc` es recomendado por Gradle para encapsular la lógica compleja de compilación de cuando se tienen submódulos. Las implementaciones de complementos y tareas personalizadas no deben vivir en el script de compilación. Es muy conveniente usar `buildSrc` siempre y cuando no sea necesario compartir el código entre varios proyectos independientes (GradleInc, 2023b). Este proyecto genérico posee 3 archivos distintos:

- **Convenciones de aplicación para java**: define un complemento que aplica configuraciones y dependencias compartidas para todos los módulos de Gradle de tipo aplicación.
- **Convenciones de librería para java**: define un complemento que aplica configuraciones y dependencias compartidas para todos los módulos de Gradle de tipo librería.
- **Convenciones comunes para java**: define un complemento que aplica configuraciones y dependencias compartidas para todos los módulos de Gradle de tipo librería y aplicación.

4.1.5. Módulo análisis de extensiones obsoletas *deprecated-extensions-analysis*

El módulo `deprecated-extensions-analysis` es de tipo *librería*. Cumple con la función de implementar la fase de la estrategia 3.1.1. En la Figura **4-7** se observa que el módulo depende de `pmd-commons` con el fin de importar la librería `pmd-core` e implementar la interfaz `Analyzer`. Se incluyen las librerías mencionadas en la Tabla **4-1**, sumada a la dependencia `pmd-xml`. `Pmd-xml` le indica a PMD que se van a analizar archivos de tipo `XML` durante su ejecución.

La clase `DeprecatedExtensionAnalyzer` se encarga de implementar la interfaz `Analyzer` (Figura **4-8**). En esta clase se referencia el archivo `deprecated-extensions-analysis/src/main/resources/rulesets/xml/basic.xml` cuyo objetivo es el de especificar la regla de búsqueda de extensiones obsoletas mediante la notación XPath en los archivos `extension-info` (Sección 3.1.1.2) y `local-extension` (Sección 3.1.1.1).

El bloque de código 4.1 denota como violación a cualquier elemento XML llamado `requires-extension` cuyo valor del atributo `name` sea el de una extensión obsoleta. Por otra parte, el bloque de código 4.2

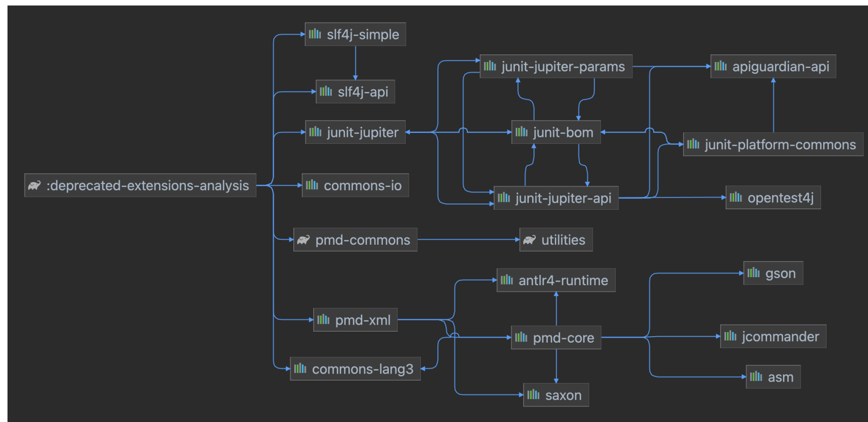

Figura **4-7**.: Diagrama de dependencias de Gradle del módulo *deprecated-extensions=analysis*.

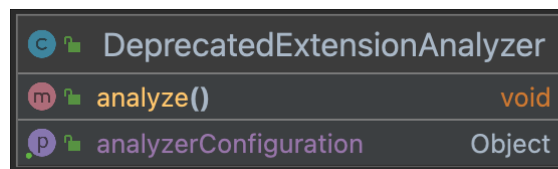

Figura **4-8**.: Diagrama de clases del módulo *deprecated-extensions-analysis*.

tiene la misma funcionalidad que la regla anterior, no obstante, busca los elementos de `xml` llamados `extension`.

```
1 <![CDATA[
2 //requires-extension[@name='acceleratorstorefrontcommons' or
3 @name='acceleratorwebservicesaddon' or
4 @name='accountsummaryaddon'
5 ...
6 ...
7 @name='yoccaddon' or
8 @name='ysapproductconfigaddon']
9 ]]>
```

Bloque de código 4.1: Notación XPath para búsqueda de extensiones obsoletas en el archivo extensionsinfo.xml.

```
1 <![CDATA[
2 //extension[@name='acceleratorstorefrontcommons' or
3 @name='acceleratorwebservicesaddon' or
```

```
4 @name='accountsummaryaddon'
5 ...
6 ...
7 @name='yoccaddon' or
8 @name='ysapproductconfigaddon']
9 ]]>
```

Bloque de código 4.2: Notación XPath para búsqueda de extensiones obsoletas en el archivo localextensions.xml.

4.1.6. Módulo análisis de incompatibilidades de SAP Commerce *commerce-incompatibilities-analysis*

El módulo `commerce-incompatibilities-analysis` es de tipo *librería*. Cumple con la función de implementar la fase de la estrategia 3.1.2. En la Figura **4-9** se observa que el módulo depende de `pmd-commons` con el fin de importar la librería `pmd-core` e implementar la interfaz `Analyzer`.

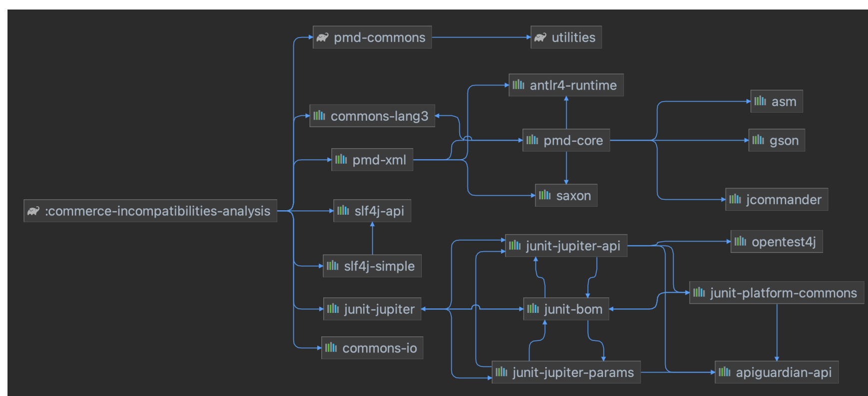

Figura **4-9**.: Diagrama de dependencias de Gradle del módulo *commerce-incompatibilities-analysis*.

La clase `CommerceIncompatibilitiesAnalyzer` se encarga de leer las propiedades manejadas `commerce-incompatibilities-analysis/src/main/resources/commerce.managed.properties` (Figura **4-10**). Estas propiedades son configuraciones de infraestructura que son administradas por SAP y no deben establecerse para la nube. Las propiedades manejadas se buscan en los archivos de propiedades de java `*.properties`, en caso de encontrar alguna se reportará el problema. El análisis se hace utilizando

las funciones comunes de java, debido a que los archivos de propiedades de java no están soportados por PMD (PMD, 2022).

Figura **4-10**.: Diagrama de clases del módulo *commerce-incompatibilties-analysis*.

Adicionalmente, como se menciona en la estrategia (Sección 3.1.2) para estas propiedades se hace la búsqueda de las palabras clave: `secret`, `password`, `key`, `pass`. Lo anterior con el fin de evitar el uso de propiedades sensibles en el código fuente como lo recomienda la documentación de SAP (SAP Sensitive configuration, 2023).

4.1.7. Módulo análisis de librerías de terceros *third-party-libraries-analysis*

El módulo `third-party-libraries-analysis` es de tipo *librería*. Cumple con la función de implementar la fase de la estrategia 3.1.3. En la Figura **4-11** se observa que el módulo depende de `pmd-commons` con el fin de importar la librería `pmd-core` e implementar la interfaz `Analyzer`.

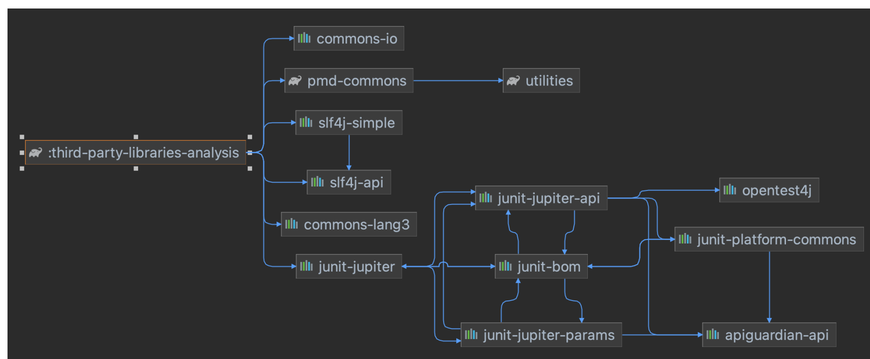

Figura **4-11**.: Diagrama de dependencias de Gradle del módulo *third-party-libraries-analysis*.

La clase `ThirdPartyLibrariesAnalyzer` se encarga de buscar las librerías `jar` que se encuentran en el código fuente de la implementación de SAP Commerce (Figura **4-12**). Posteriormente lee el manifesto

de estos `jar` con el fin de obtener el nombre de la librería y el número de la versión. El nombre de la librería es buscado en el repositorio central de maven central (SonatypeInc, 2023) y se contrasta la versión de la librería junto con la última versión encontrada. En caso que la versión de maven-central sea mayor se reportará la violación.

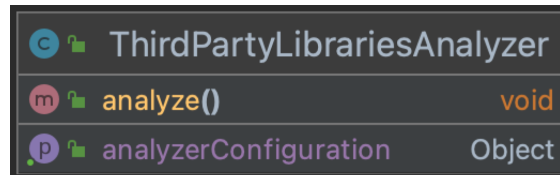

Figura **4-12**.: Diagrama de clases del módulo *third-party-libraries-analysis*.

4.1.8. Módulo análisis de palabras clave de motores de base de datos *database-keyword-commerce-analysis*

El módulo `database-keyword-commerce-analysis` es de tipo *librería*. Cumple con la función de implementar la fase de la estrategia 3.1.4. En la Figura **4-13** se observa que el módulo depende de `pmd-commons` con el fin de importar la librería `pmd-core` e implementar la interfaz `Analyzer`.

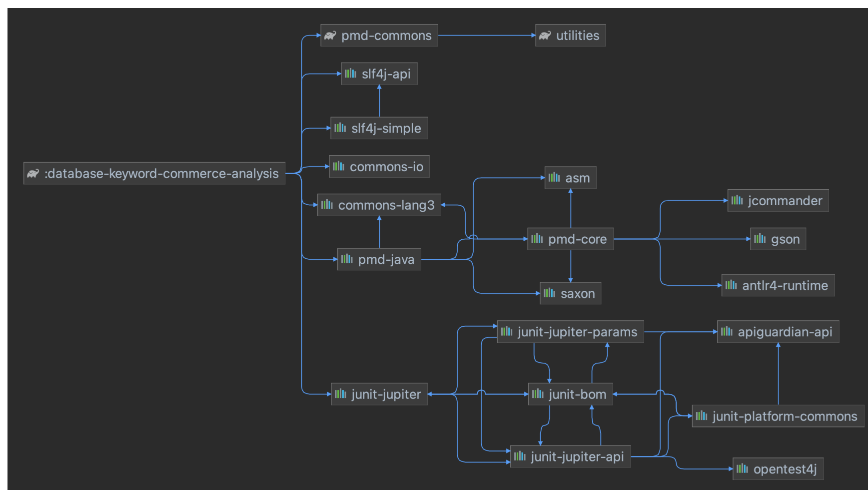

Figura **4-13**.: Diagrama de dependencias de Gradle del módulo *database-keyword-commerce-analysis*.

La Figura **4-14** contiene tres clases que cumplen con las siguientes funciones:

- `DatabaseKeywordCommerceAnalyzer`: Implementa la interfaz `Analyzer` del módulo `pmd-commons` que cumple con el fin de iniciar la detección de palabras clave de base de datos. Esta clase inicializa la herramienta PMD para indicar que se van a buscar palabras clave en los archivos de tipo `*.java`.
- `DatabaseKeywordData`: Clase de datos que posee dos propiedades: `keywordGroup` contiene el nombre del motor de la base de datos que se está buscando; `keywords` almacena las palabras clave del motor de base de datos que se van a buscar.
- `DatabaseKeywordRule`: Esta clase extiende la clase de la librería PMD `AbstractJavaRule`, la cual define el método `visit`, en este caso se revisa si el objeto `ASTLiteral` es de tipo `String` y contiene alguna palabra clave del motor de base de datos.

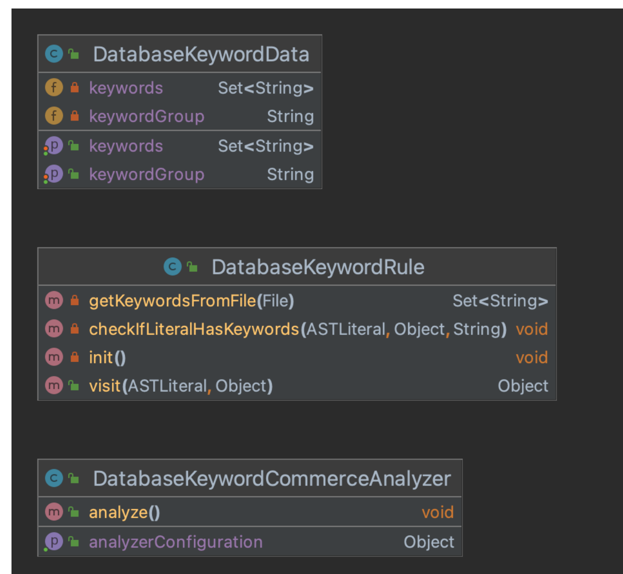

Figura **4-14**.: Diagrama de clases del módulo *database-keyword-commerce-analysis*.

4.1.9. Módulo análisis de código java *java-commerce-analysis*

El módulo `java-commerce-analysis` es de tipo *librería*. Cumple con la función de implementar la fase de la estrategia 3.2. En la Figura **4-15** se observa que el módulo depende de `pmd-commons` con el fin de importar la librería `pmd-core` e implementar la interfaz `Analyzer`. Adicionalmente, implementa la

librería `pmd-java` cuyo objetivo es el de utilizar el conjunto de reglas por defecto para java que tiene la herramienta `PMD`.

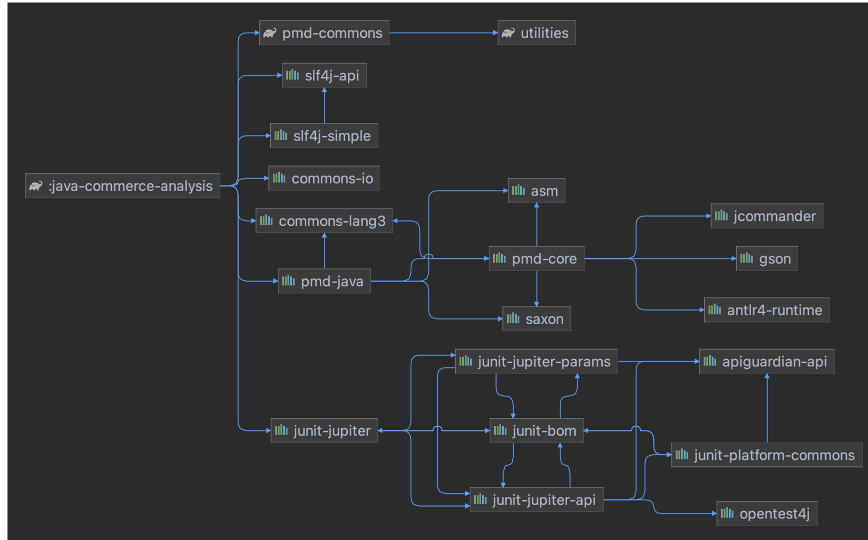

Figura **4-15**.: Diagrama de dependencias de Gradle del módulo *java-commerce-analysis*.

La Figura **4-16** contiene la clase `JavaCommerceAnalyzer`, la cual simplemente se encarga de configurar y ejecutar el conjunto de reglas por defecto de PMD, que se encuentra en su repositorio público de GitHub `https://github.com/pmd/pmd/blob/master/pmd-java/src/main/resources/rulesets/java/basic.xml`. Estas reglas se encuentran documentadas con detalle en la Sección 3.2.

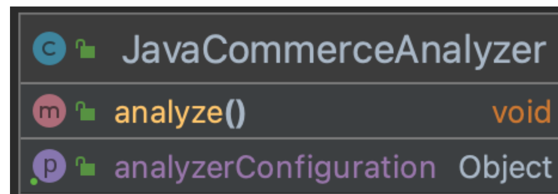

Figura **4-16**.: Diagrama de clases del módulo *java-commerce-analysis*.

4.1.10. Módulo análisis de palabras clave de motores de base de datos *javascript-commerce-analysis*

El módulo `javascript-commerce-analysis` es de tipo *librería*. Cumple con la función de implementar la fase de la estrategia 3.3. En la Figura **4-17** se observa que el módulo depende de `pmd-commons` con el

fin de importar la librería `pmd-core` e implementar la clase `Analyzer`. Adicionalmente, implementa la librería `pmd-javascript` cuyo objetivo es el de utilizar el conjunto de reglas por defecto para javascript que tiene la herramienta `PMD`.

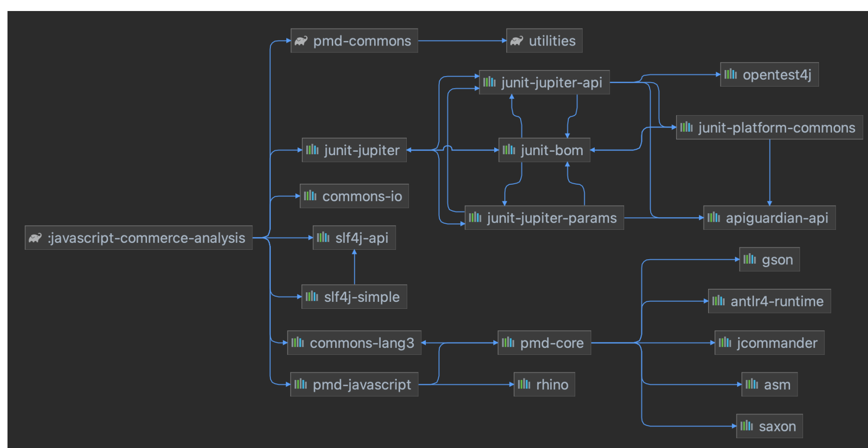

Figura **4-17**.: Diagrama de dependencias de Gradle del módulo *javascript-commerce-analysis*.

La Figura **4-18** contiene la clase `JavascriptCommerceAnalyzer`, la cual se encarga de configurar y ejecutar el conjunto de reglas por defecto de PMD, que se encuentra en su repositorio público de GitHub `https://github.com/pmd/pmd/blob/master/pmd-javascript/src/main/resources/rulesets/ecmascript/basic.xml`. Estas reglas se encuentran documentadas con detalle en la Sección 3.3.

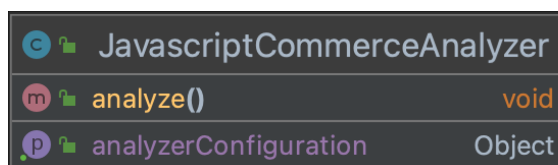

Figura **4-18**.: Diagrama de clases del módulo *javascript-commerce-analysis*.

4.2. Flujo de ejecución de la herramienta

Esta sección se encarga de explicar el flujo de ejecución de la herramienta el cual comprende tres fases secuenciales: instalación y ejecución (Sección 4.2.1), inicialización (Sección 4.2.2) y análisis (Sección 4.2.3). A continuación, se explica cada una en detalle.

4.2.1. Instalación y ejecución

La primera fase del flujo es la de instalación y ejecución, la cual es pre-requisito para las fases posteriores de inicialización y análisis. Las instrucciones y requisitos técnicos para ejecutar el proyecto también se pueden encontrar en el repositorio público de GitHub: `https://github.com/roger8849/sap_commerce_code_analyzer`.
La herramienta requiere ser ejecutada en dispositivos con las siguientes características técnicas:

- **Sistema operativo (obligatorio):**
  - **Windows:** 10, 11, Server 2022 Datacenter (Google cloud).
  - **Linux:** Ubuntu 20.04 o superior.
  - **Mac Os X:** Ventura 13.4
  - **Otros:** Es altamente probable que en otros sistemas operativos y/o versiones también funcione la herramienta, no obstante, no se tiene evidencia de ello. Es requerido que el sistema operativo soporte OpenJDK version 11 o superior.
- **Lenguaje de programación Java (obligatorio):** Se requiere que OpenJDK versión 11 -19 se encuentre instalada. El API estándar de OpenJDK fue usado, por lo tanto, es posible usar versiones comerciales de Java.
- **Permisos de escritura en la carpeta raíz del usuario del sistema operativo (obligatorio):** Los reportes generados por la herramienta se escriben en la carpeta `$HOME/reports/`[2]. Es necesario que el usuario tenga permiso de escritura en su directorio raíz.
- **Gradle (opcional):** El repositorio `https://github.com/roger8849/sap_commerce_code_analyzer`contiene Gradle wrapper. Esto permite ejecutar el proyecto mediante un trabajo de Gradle sin necesidad de instalarlo. No obstante, si se desea extender su funcionamiento, es necesaria la instalación de Gradle.
- **Git (opcional):** Es requerido el uso de git en caso de querer clonar el código fuente de la herramienta desde el repositorio público de GitHub.

[2] La variable de ambiente `$HOME` se refiere al directorio base del usuario en el sistema operativo.

Las demás librerías y herramientas mencionadas ya se encuentran empaquetas en los binarios de la herramienta. Una vez los requisitos técnicos se encuentren instalados, hay dos opciones de ejecución mediante Gradle wrapper o un IDE, las cuales se explican a continuación:

- **Ejecución mediante Gradle usando:** Descargar código fuente usando alguna de las siguientes opciones:
    1. Clonar la rama `main` usando el comando de Git: `git clone https://github.com/roger8849/sap_commerce_code_analyzer .`
    2. Descargar el archivo zip correspondiente a la rama `main` desde: `https://github.com/roger8849/sap_commerce_code_analyzer/archive/refs/heads/main.zip`

  Posteriormente, se debe ejecutar alguno de los siguientes comandos en la terminal:
    - Windows: `gradlew.bat run`
    - Unix (Linux y MacOS): `./gradlew run`
- **Ejecución mediante IDE:** Si se desea ejecutar mediante un IDE debe descargarse el código fuente e importar el proyecto con un IDE que sea compatible con java y Gradle. Posteriormente, se debe configurar el perfil de ejecución del trabajo de gradle `app`, siguiendo las instrucciones de cada IDE. En la Figura **4-19** se muestra una configuración de referencia usando la herramienta IntelliJ IDEA.

En las secciones posteriores se detallan las siguientes fases de inicialización (Sección 4.2.2) y análisis (Sección 4.2.3) que hacen parte del flujo de ejecución del proyecto.

4.2.2. Inicialización

La fase de inicialización tiene como objetivo leer los parámetros y configuraciones que se utilizarán en la fase de análisis. La Figura **4-20** contiene el diagrama de flujo de la fase de inicialización de la herramienta, el cual comprende los siguientes elementos:

1. **Leer ubicación de código fuente:** Esta tarea se encarga de obtener la ruta en el sistema de archivos del sistema operativo, que contiene el código fuente que se va a analizar. Esta tarea comprende las siguientes situaciones:

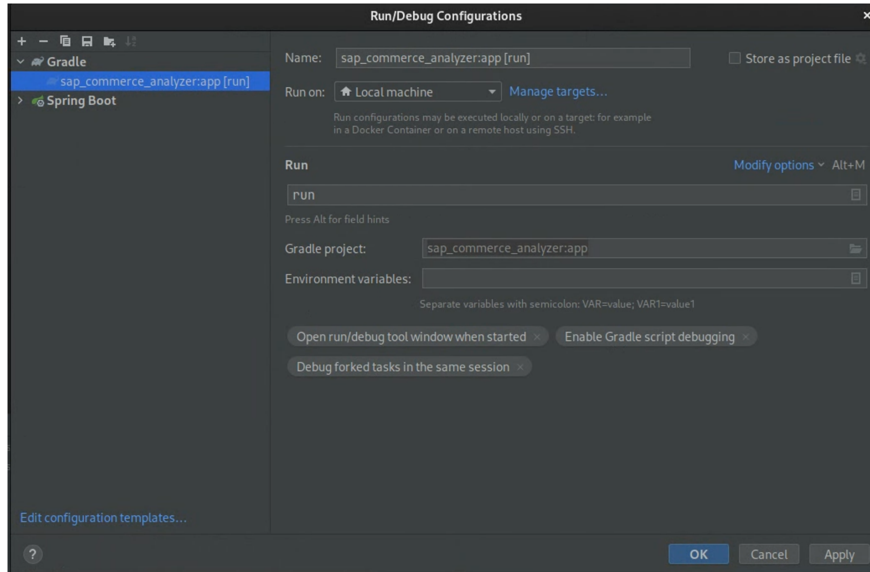

Figura **4-19**.: Configuración de ejecución de referencia de la herramienta IntelliJ IDEA.

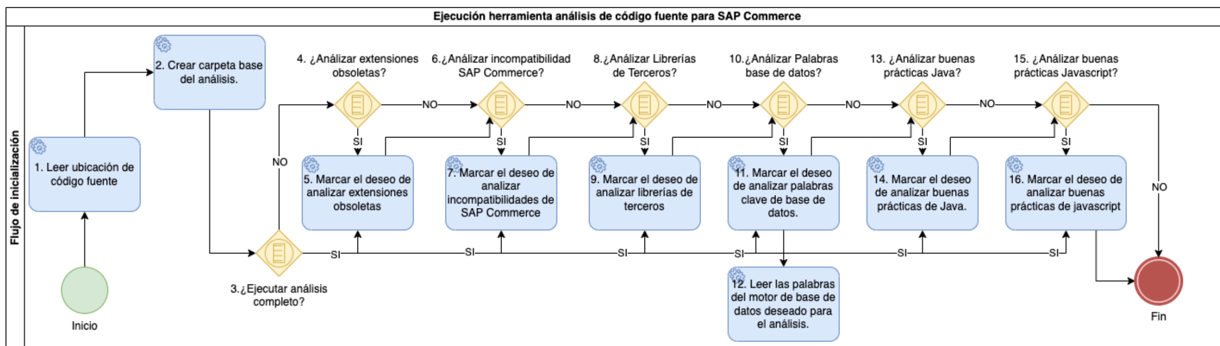

Figura **4-20**.: Diagrama de flujo de la fase de inicialización.

- **Variable de ambiente `HYBRIS_HOME`:** La herramienta primero intenta ver si la variable de ambiente de sistema operativo `HYBRIS_HOME` se encuentra configurada. Si la variable existe, se le pregunta al usuario si desea usar el valor de esta variable como ubicación del código fuente. Si la variable no existe o el usuario no quiere usarla, se lee la variable desde consola.
- **Leer ruta desde consola:** Si la variable de ambiente de sistema operativo `HYBRIS_HOME` no existe o el usuario no la quiere usar, entonces se le pide la ruta absoluta en el sistema de archivos donde se encuentra el código a analizar. Esta ruta debe ser alcanzable por el programa, de lo contrario, el programa terminará con un error.

2. **Crear carpeta base del análisis:** Tarea automática que ejecuta las siguientes acciones:
   - Crea la carpeta `$HOME/reports` en caso de no existir.
   - Crea la carpeta `$HOME/reports/report-<yyyymmddHHMMSS>`, donde `yyyy` es el año, `mm` es el mes, `dd` es el día, `HH` es la hora, `MM` son los minutos, `SS` son los segundos correspondientes al tiempo de la ejecución. Esta carpeta se crea cada vez que el usuario ejecute un nuevo análisis.

   En esas carpetas se guardarán los archivos con los resultados de ejecución de cada análisis.

3. **¿Ejecutar análisis completo?** Indaga y lee la respuesta del usuario para determinar si desea hacer un análisis completo o no. El usuario puede responder:
   - **Si:** se ejecutan las tareas 5, 7, 9, 11, 14 y 16 de este flujo.
   - **No:** se ejecutan las tareas 4, 6, 8, 10, 13 y 15 de este flujo.

4. **¿Analizar extensiones obsoletas análisis completo?** Indaga sobre el deseo del usuario de ejecutar el análisis de extensiones obsoletas.
   - **Si:** se ejecuta la tarea 5.
   - **No:** se pasa a la tarea 6.

5. **Marcar el deseo de analizar extensiones obsoletas:** Asigna el valor de verdadero (`true`) a la variable booleana que se leerá durante la fase de análisis.

6. **¿Analizar incompatibilidades SAP Commerce?** Indaga sobre el deseo del usuario de ejecutar el análisis de incompatibilidades de SAP Commerce.
   - **Si:** se ejecuta la tarea 7.
   - **No:** se pasa a la tarea 8.
7. **Marcar el deseo de analizar incompatibilidades de SAP Commerce:** Asigna el valor de verdadero (`true`) a la variable booleana que se leerá durante la fase de análisis.
8. **¿Analizar librerías de terceros?** Indaga sobre el deseo del usuario de ejecutar el análisis de librerías de terceros.
   - **Si:** se ejecuta la tarea 9.
   - **No:** se pasa a la tarea 10.
9. **Marcar el deseo de analizar librerías de terceros:** Asigna el valor de verdadero (`true`) a la variable booleana que se leerá durante la fase de análisis.
10. **¿Analizar palabras clave de base de datos?** Indaga sobre el deseo del usuario de ejecutar el análisis de palabras clave de base de datos.
    - **Si:** se ejecuta la tarea 11.
    - **No:** se pasa a la tarea 13.
11. **Marcar el deseo de analizar palabras clave de base de datos:** Asigna el valor de verdadero (`true`) a la variable booleana que se leerá durante la fase de análisis.
12. **Leer las palabras clave del motor de base de datos a analizar:** Pregunta al usuario cuál motor de base de datos debe usarse como base para buscar por sus palabras clave. A la fecha de escritura de este documento (20 de octubre de 2023) se soportan los siguientes motores de base de datos: mysql, oracle, postgresql, sap hana, sql server.
13. **¿Analizar buenas prácticas de Java?** Indaga sobre el deseo del usuario de ejecutar el análisis de buenas prácticas de Java.
    - **Si:** se ejecuta la tarea 14.

- **No:** se pasa a la tarea 15.

14. **Marcar el deseo de analizar buenas prácticas de java:** Asigna el valor de verdadero (`true`) a la variable booleana que se leerá durante la fase de análisis.

15. **¿Analizar buenas prácticas de Javascript?** Indaga sobre el deseo del usuario de ejecutar el análisis de buenas prácticas de Javascript.

    - **Si:** se ejecuta la tarea 16.
    - **No:** la fase de inicialización termina.

16. **Marcar el deseo de analizar buenas prácticas de Javascript:** Asigna el valor de verdadero (`true`) a la variable booleana que se leerá durante la fase de análisis.

Una vez la fase de inicialización termina, la herramienta inicia el proceso de análisis detallado en la siguiente sección.

4.2.3. Análisis

Una vez el proyecto ha sido ejecutado y los parámetros de inicialización han sido leídos , se inicia la fase de análisis. La Figura **4-21** contiene el diagrama de flujo de la fase de análisis de la herramienta, el cual comprende los siguientes elementos:

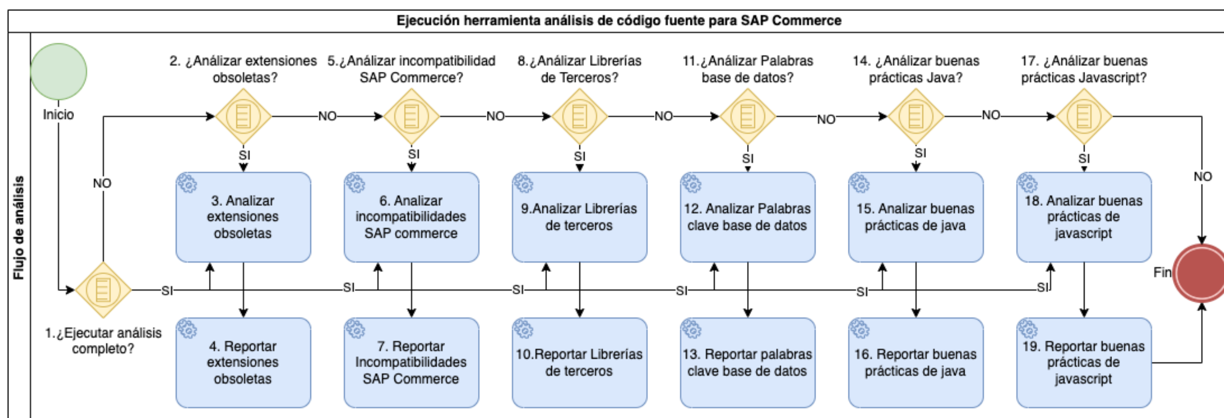

Figura **4-21**.: Diagrama de flujo de la fase de análisis en la ejecución.

1. **¿Ejecutar análisis completo?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar o no un análisis completo. El condicional comprende las siguientes opciones:
   - **Si:** En caso que el usuario desee ejecutar un análisis completo se ejecutan las tareas 3, 6, 9, 12, 15 y 18 del flujo.
   - **No:** Si el usuario desea ejecutar sólo ciertas fases del análisis se pasa a la tarea 2.
2. **¿Analizar extensiones obsoletas?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar el análisis de extensiones obsoletas. El condicional comprende las siguientes opciones:
   - **Si:** En caso que el usuario desee ejecutar el análisis de extensiones obsoletas se pasa a la tarea 3.
   - **No:** si el usuario no desea ejecutar este análisis, entonces se pasa a la tarea 5.
3. **Analizar extensiones obsoletas:** Inicia el módulo 4.1.5 para detectar el uso de extensiones obsoletas en el código fuente.
4. **Reportar extensiones obsoletas:** El módulo 4.1.5 escribe en el archivo `./$HOME/reports/report-<yyyymmddHHMMSS>/deprecated_extensions_report.csv`, en el que se reporta el uso de extensiones obsoletas.
5. **¿Analizar incompatibilidades de SAP Commerce?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar el análisis de incompatibilidades de SAP Commerce. El condicional comprende las siguientes opciones:
   - **Si:** En caso que el usuario desee ejecutar el análisis de incompatibilidades de SAP Commerce se pasa a la tarea 6.
   - **No:** si el usuario no desea ejecutar este análisis entonces se pasa a la tarea 8.
6. **Analizar incompatibilidades SAP Commerce:** Inicia el módulo 4.1.6 para detectar el uso de características de SAP Commerce incompatibles con la arquitectura en la nube.

7. **Reportar incompatibilidades de SAP Commerce:** El módulo 4.1.6 escribe en el archivo `./$HOME/reports/report-<yyyymmddHHMMSS>/incompatibilities_report.csv`, en el que se reporta el uso de incompatibilidades de SAP Commerce.

8. **¿Analizar librerías de terceros?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar el análisis de librerías de terceros. El condicional comprende las siguientes opciones:
   - **Si:** En caso que el usuario desee ejecutar el análisis de librerías de terceros se pasa a la tarea 9.
   - **No:** si el usuario no desea ejecutar este análisis entonces se pasa a la tarea 11.

9. **Analizar librerías de terceros:** Inicia el módulo 4.1.7 para analizar la versión de las librerías de terceros.

10. **Reportar análisis de librerías de terceros:** El módulo 4.1.7 escribe en el archivo `./$HOME/reports/report-<yyyymmddHHMMSS>/third_party_libraries_report.csv`, en el que se reporta las versiones desactualizadas de las librerías de terceros.

11. **¿Analizar palabras clave de base de datos?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar el análisis de palabras clave de base de datos. El condicional comprende las siguientes opciones:
    - **Si:** En caso que el usuario desee ejecutar el análisis de palabras clave de base de datos se pasa a la tarea 12.
    - **No:** si el usuario no desea ejecutar este análisis, entonces se pasa a la tarea 14.

12. **Analizar palabras clave de base de datos:** Inicia el módulo 4.1.8 para analizar el análisis de palabras clave de base de datos.

13. **Reportar análisis de librerías de terceros:** El módulo 4.1.8 escribe en el archivo `./$HOME/reports/report-<yyyymmddHHMMSS>/database_report.csv`, en el que se reporta las funciones clave de base de datos en el código fuente.

14. **¿Analizar buenas prácticas de java?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar el análisis buenas prácticas de java. El condicional comprende las siguientes opciones:
    - **Si:** En caso que el usuario desee ejecutar el análisis de buenas prácticas de java se pasa a la tarea 15.
    - **No:** si el usuario no desea ejecutar este análisis, entonces se pasa a la tarea 17.
15. **Analizar buenas prácticas de java:** Inicia el módulo 4.1.9 para analizar las buenas prácticas de java recomendadas por la librería PMD (PMD, 2022).
16. **Reportar análisis de librerías de terceros:** El módulo 4.1.9 escribe en el archivo `./$HOME/reports/report-<yyyymmddHHMMSS>/java_report.csv`, en el que se reporta los hallazgos en torno a las buenas prácticas de java.
17. **¿Analizar buenas prácticas de javascript?**: Lee la variable obtenida en la fase de inicialización para determinar si el usuario desea ejecutar el análisis buenas prácticas de javascript. El condicional comprende las siguientes opciones:
    - **Si:** En caso que el usuario desee ejecutar el análisis de buenas prácticas de javascript se pasa a la tarea 15.
    - **No:** si el usuario no desea ejecutar este análisis, entonces el programa finaliza.
18. **Analizar buenas prácticas de javascript:** Inicia el módulo 4.1.10 para analizar las buenas prácticas de javascript recomendadas por la librería PMD (PMD, 2022).
19. **Reportar análisis de librerías de terceros:** El módulo 4.1.10 escribe en el archivo `./$HOME/reports/report-<yyyymmddHHMMSS>/javascript_report.csv`, en el que se reporta los hallazgos en torno a las buenas prácticas de javascript. Posterior a esta tarea el programa finaliza.

5. Evaluación de la estrategia mediante estudios de caso de SAP Commerce

En los capítulos anteriores se presentó la estrategia de análisis de código fuente en SAP Commerce junto con su respectiva implementación. Se estableció un enfoque sólido para examinar el código fuente en busca de problemas potenciales, a la hora de migrar SAP Commerce a la nueva arquitectura en la nube. Este capítulo marca la transición hacia la siguiente fase del trabajo, que se centra en la evaluación de la estrategia propuesta. El objetivo principal de esta fase es llevar a cabo pruebas exhaustivas sobre implementaciones reales de SAP Commerce, con el fin de validar la efectividad y el correcto funcionamiento de la herramienta de análisis.

La fase de evaluación es crucial para garantizar la calidad y el buen funcionamiento de la herramienta de análisis automático de SAP Commerce. Proporciona información valiosa para el equipo de desarrollo, permitiendo corregir problemas identificados, así acelerar el proceso de estimación y desarrollo de una migración de SAP Commerce a la nube.

La evaluación se realiza mediante la aplicación de pruebas sobre el código fuente del sistema. Estas pruebas se llevan a cabo para tres distintas implementaciones de SAP Commerce. En la Sección 5.1 se explican los elementos que componen el reporte de hallazgos encontrados por la herramienta. En las secciones posteriores se presentan los resultados obtenidos tras las pruebas sobre los tres estudios de caso: Código tutorial de SAP Commerce "*Commerce 123*" (Sección 5.2), Implementación real de una tienda de venta comercio a cliente (B2C) de un megamercado en latinoamérica (Sección 5.3), y por último, una implementación real de una tienda de venta de comercio a comercio (B2B) de un distribuidor de artículos e insumos electrónicos para Estados unidos, Canadá y Europa Occidental (Sección 5.4).

El listado completo de hallazgos encontrados por la herramienta para los tres proyectos se encuentra disponible en el repositorio público de GitHub. Puede acceder al listado completo de hallazgos en el siguiente enlace: `https://github.com/roger8849/sap_commerce_code_analyzer/tree/main/evaluation_results`.

5.1. Elementos de los reportes de evaluación

En la Sección 4.2.3, correspondiente a la fase de análisis, se mencionó que el resultado de cada uno de los análisis estará en formato CSV (Comma-Separated Values). Estos archivos CSV contienen columnas relacionadas que proporcionan información detallada sobre el análisis realizado. A continuación, se describen las columnas que se encuentran en estos archivos:

- **Paquete (Package):** Contiene el paquete de java de la clase que contiene la violación. En caso de que el archivo no sea de tipo Java, entonces su valor estará vacío.
- **Archivo (File):** Ruta del sistema de archivos donde se encuentra el archivo con la violación.
- **Prioridad (Priority):** Valor numérico que representa una de las siguientes prioridades:
    1. Alta: El cambio es absolutamente requerido
    2. Media-alta: El cambio es altamente recomendado.
    3. Media: El cambio es recomendado
    4. Media-baja: El cambio es opcional
    5. Baja:. El cambio es altamente opcional
- **Línea (Line):** Número de línea donde se encuentra la violación dentro del archivo.
- **Descripción (Description):** Descripción que contiene la regla en la que se explica por qué se encontró una violación.
- **Conjunto de reglas (Rule set):** Conjunto de reglas que contiene la violación.
- **Regla (Rule):** Regla específica dentro del conjunto de reglas que reporta la violación.

Cada columna proporciona información valiosa para comprender los problemas identificados y tomar las medidas adecuadas para su corrección de acuerdo a la prioridad.

5.2. Estudio de caso 1: SAP Commerce 123

Este estudio de caso comprende código fuente analizado correspondiente a una implementación que SAP Commerce ofrece como un proyecto de ejemplo en su documentación. Está diseñado para brindar capacitación y orientación a los desarrolladores. Este proyecto de ejemplo se conoce como "*SAP Commerce 123*".

El proyecto "*SAP Commerce 123*" se basa en una implementación personalizada de una tienda en línea para la venta de tiquetes de conciertos de música. Su objetivo es proporcionar un escenario de ejemplo que demuestre cómo se pueden personalizar y adaptar las funcionalidades de SAP Commerce para satisfacer las necesidades específicas de una tienda en línea de este tipo.

Cabe destacar que la implementación personalizada sobre SAP Commerce se ha realizado utilizando el proyecto de ejemplo "*SAP Commerce 123*" como punto de partida. Sin embargo, la evaluación de escenarios reales implementados por desarrolladores de SAP Commerce se explora en las Secciones 5.3 y 5.4, respectivamente.

El código fuente evaluado del proyecto "*SAP Commerce 123*" posee las siguientes características:

- **Ubicación del código fuente evaluado:** `https://github.com/roger8849/sap_commerce_code_analyzer/tree/main/commerce_123`
- **Versión de Java:** Open JDK 11.
- **Versión de SAP Commerce:** 2105.
- **Motor de base de datos:** MySQL.
- **Archivos de Java:** 47.
- **Archivos de Javascript:** 0.
- **Archivos de propiedades de java:** 18.

- **Archivos XML:** 21.

La herramienta se ejecutó con los parámetros que se observan en el bloque de código 5.1, En donde se proporcionó la ruta absoluta donde se encuentra el código fuente que se va a analizar, se especificó que se desea hacer un análisis completo de todas las secciones, y por último, se pidió buscar palabras clave del motor de base de datos MySQL.

```
1  # Comando de ejecución de programa.
2  rramirezespejo-macbookpro2:sap_commerce_analyzer rramirezespejo$ ./gradlew run
3  Starting a Gradle Daemon (subsequent builds will be faster)
4  > Task :app:run
5  [main] INFO co.edu.unal.utilities.ConsoleUtils -
       -----------------------------------------------
6  [main] INFO co.edu.unal.utilities.ConsoleUtils - - Welcome to SAP Commerce
       migration analyzer. -
7  [main] INFO co.edu.unal.utilities.ConsoleUtils -
       -----------------------------------------------
8  ...
9  ...
10 [main] INFO co.edu.unal.utilities.InitUtils - Please enter the absolute path
       of the hybris dir to analyze:
11 # Ubicación en el sistema de archivos de Commerce 123
12 /Users/rrammirezespejo/dev_home/GitHub/sap_commerce_static_code_analyzer/
       commerce_123
13 [main] INFO co.edu.unal.utilities.ConsoleUtils - Do you want to do a complete
       analysis?
14 (yes). no.
15 # No se ingresó ningún valor para usar (yes) como defecto
16
17 ...
18 ...
19 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Which database keywords do
       you want to detect. -
20 [main] INFO co.edu.unal.utilities.ConsoleUtils -
       -------------------------------------------------
21 [main] INFO co.edu.unal.utilities.ConsoleUtils -
```

```
22 1. postgresql.
23 2. oracle.
24 3. mysql.
25 4. sqlserver.
26 5. saphana.
27 # Se ingresó  el número 3 para analizar palabras clave del motor de base de
       datos MySQL.
28 3
29 ...
30 ...
31 ...
32 [main] INFO co.edu.unal.utilities.ConsoleUtils -
       -------------------------------------------------------------
33 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Analysis ended. Results of
       the analysis were written in: -
34 [main] INFO co.edu.unal.utilities.ConsoleUtils -
       -------------------------------------------------------------
35 # Ruta en el sistema de archivos en donde se escribieron los resultados del
       análisis:
36
37 [main] INFO co.edu.unal.app.App - /Users/rramirezespejo/reports/analysis-
       report-20230927150941/
```

Bloque de código 5.1: Parámetros de entrada evaluación en Commerce 123.

Una vez que se ha indicado que el motor de base de datos a analizar es MySQL, se inicia la ejecución del análisis. La Tabla **5-1** proporciona un resumen de los hallazgos encontrados, con un ejemplo de cada tipo de hallazgo.

Tabla **5-1**.: Resumen de resultados del estudio de caso 1: SAP Commerce 123.

Paquete	Archivo	Prioridad	Línea	Descripción	Conjunto de Reglas	Regla
Extensiones obsoletas						
No hay resultados						
Incompatibilidades de SAP Commerce						

	/commerce_123 /config/local.properties	2	10	Property 'mail.smtp.user' contains the key '.user' which seems to contain sensitive information	Commerce Incompatibilities	Validate Sensitive Key
	/commerce_123 /config/local.properties	2	11	Property 'mail.smtp.password' contains the key '.password' which seems to contain sensitive information	Commerce Incompatibilities	Validate Sensitive Key
Librerías de terceros						
	/commerce_123/bin /custom/concerttours /web/webroot /WEB-INF/lib /displaytag-1.2.jar	5		Jar with name displaytag-1.2 couldn't find the version in its manifest.	Third party libraries	Jar Version Not In Jar Manifest
	/commerce_123 /config/licence/hybrislicence.jar	5		Jar with name hybrislicence couldn't find the version in its manifest.	Third party libraries	Jar Version Not In Jar Manifest
Palabras clave de base de datos						
concerttours. daos.impl	/commerce_123/bin /custom/concerttours /src/concerttours /daos/impl/DefaultNewsDAO.java	1	32	String literal "WHERE {date} >= DATE" contains the DATETIME function 'DATE' which belongs to the database engine MYSQL.	Database keyword rule	Database Keyword Rule
concerttours. daos.impl	/commerce_123/bin /custom/concerttours /src/concerttours /daos/impl/DefaultNewsDAO.java	1	33	String literal "AND date <= DATE" contains the DATETIME function 'DATE' which belongs to the database engine MYSQL.	Database keyword rule	Database Keyword Rule
Buenas prácticas de Java						

concerttours .constraints	/commerce_123/bin /custom/concert-tours /src/concert-tours /constraints /NotLoremIpsumValidator.java	3	16	When doing a String.toLowerCase() / toUpperCase() call use a Locale	Error Prone	Use Locale With Case Conversions
concerttours. controller	/commerce_123/bin /custom/concert-tours /src/concert-tours /controller /BandController.java	3	21	Avoid the use of value in annotations when its the only element	Code Style	Unnecessary Annotation Value Element
concerttours. facades	/commerce_123/bin /custom/concert-tours /src/concert-tours /facades/TourFacade.java	1	6	Final parameter in abstract method	Code Style	Final Parameter In Abstract Method
concerttours. events	/commerce_123/bin /custom/concert-tours /src/concert-tours /events/BandAlbumSalesEvent.java	4	38	Useless parentheses.	Code Style	Useless Parentheses
concerttours. jalo	/commerce_123/bin /custom/concert-tours /testsrc/concerttours /jalo /ConcerttoursTest.java	4	7	Unused import ’org.junit.Assert.assertTrue’	Code Style	Unnecessary Import

concerttours. facades.impl	/commerce_123/bin /custom/concert- tours /testsrc/con- certtours /facade- s/impl /Default Band Facade Integration WithProperties- Test.java	4	4	Unnecessary import from the java.lang package 'ja- va.lang.InterruptedException'	Code Style	Unnecessary Import
concerttours. setup	/commerce_123/bin /custom/concert- tours /src/con- certtours /setup /ConcerttoursCus- tomSetup.java	2	87	Logger calls should be su- rrounded by log level guards.	Best Practices	Guard Log Sta- tement
Buenas prácticas de Javascript						
No hay resultados						

Fuente: Elaboración propia

La evaluación realizada generó una serie de resultados significativos que arrojaron luz sobre el estado del código fuente y proporcionaron información valiosa para mejorar la calidad y el rendimiento del proyecto ”SAP Commerce 123”. A continuación se presentan algunos de los resultados obtenidos (resumen en Tabla **5-1**):

- **Total de problemas identificados:** La evaluación reveló un total de 60 problemas encontrados en el código fuente de la implementación personalizada de “SAP Commerce 123”.
- **Tipos y severidad de los problemas:** Los 60 problemas encontrados se discriminan de la siguiente forma:
  1. **Extensiones obsoletas:** 0.
  2. **Incompatibilidades de SAP Commerce:** 2 hallazgos de prioridad Media-alta.
  3. **Librerías de terceros:** 2 hallazgos de prioridad baja.

4. **Palabras clave de base de datos:** 5 hallazgos de prioridad alta.
5. **Análisis de Java:** 50 hallazgos: 3 prioridad alta, 2 prioridad media-alta, 21 prioridad media, 24 prioridad baja.
6. **Análisis de Javascript:** 0.

Los hallazgos encontrados son representativos del tamaño del repositorio y el tipo de archivos que se analizaron. Es importante destacar que, en el caso específico del proyecto “SAP Commerce 123”, no se reportó ningún hallazgo relacionado con archivos de JavaScript. Esto se debe a que la implementación personalizada de este proyecto no incluye archivos de JavaScript en su estructura de código fuente.
Por otra parte, es importante destacar que el proyecto “SAP Commerce 123”se utiliza como un ejemplo de aprendizaje para personalizar SAP Commerce. Sin embargo, durante el análisis del código Java, se identificaron 5 literales de tipo String que contienen palabras clave específicas del motor de base de datos MySQL. Esta situación presenta dos problemas significativos.
En primer lugar, el uso de palabras clave del motor de base de datos MySQL imposibilita la migración del proyecto a la nube, en particular al utilizar el motor de base de datos Azure SQL. Dado que el proyecto se basa en SAP Commerce, es recomendable adaptarlo a las mejores prácticas y estándares de la plataforma, lo que incluye la compatibilidad con diferentes motores de base de datos.
En segundo lugar, esta implementación rompe la recomendación de los consultores de SAP Commerce, que aconsejan evitar las dependencias directas al motor de base de datos. Estas dependencias directas pueden generar problemas de portabilidad y mantenibilidad del código, además de limitar la capacidad de adaptación a diferentes entornos y tecnologías.
Con base en estos hallazgos, se recomienda a la compañía SAP que corrija esta implementación en el proyecto “SAP Commerce 123”. Es fundamental evitar la incorporación de malas prácticas desde el proceso de formación de desarrolladores. Esto implica modificar los literales de tipo String que contienen palabras clave de MySQL, reemplazándolos por soluciones más genéricas y compatibles con diferentes motores de base de datos como el lenguaje de dominio específico Flexible Search.
Es importante tener en cuenta que el número bajo de problemas identificados en el análisis de buenas prácticas de Java sugiere que, en general, se ha realizado un esfuerzo razonable en la implementación. Sin embargo, es crucial abordar los problemas de prioridad alta y media-alta para garantizar una implementación sólida y preparada para futuras actualizaciones y migraciones.

Adicionalmente, se menciona que durante el análisis se encontraron únicamente 2 problemas relacionados con incompatibilidades de SAP Commerce. Esto indica que, en general, el proyecto “SAP Commerce 123”ha logrado mantener una alta compatibilidad con las funcionalidades y las recomendaciones de SAP Commerce.

Es importante señalar que el proyecto SAP Commerce 123 no hace referencia a ninguna extensión obsoleta. Esto reduce significativamente el esfuerzo requerido para migrar a la arquitectura de SAP Commerce a la nube.

En conclusión, la implementación de SAP Commerce 123 es un proyecto tutorial que no contiene muchas funcionalidades. Como resultado, no se requeriría un gran esfuerzo de adaptación para ser implementado en la arquitectura en la nube de SAP Commerce.

5.3. Estudio de caso 2: Tienda en línea macromercado

El estudio de caso analizado en esta sección se refiere al análisis del código fuente de una implementación real de SAP Commerce de un macrocomercio que comercializa una variedad de productos al por menor en un país de América Latina. Es importante destacar que la identidad de este macrocomercio, así como los hallazgos presentados, se mantienen en anonimato, ya que se obtuvo autorización verbal para realizar pruebas en el código fuente con la condición de mantener en secreto su identidad y el código fuente analizado. Con base en lo anterior, a lo largo de esta sección se hará referencia a esta empresa con el pseudónimo de “TiendaRetailB2C”. El código fuente evaluado posee las siguientes características:

- **Versión de Java:** Open JDK 8.
- **Versión de SAP Commerce:** 6.3.0.4.
- **Motor de base de datos:** Oracle 11g.
- **Total de archivos:** 2650 archivos. 315170 líneas de código.
  - **Archivos de Java:** 1115 archivos. 143134 líneas de código.
  - **Archivos de Javascript:** 234 archivos. 72560 líneas de código.

- **Archivos de propiedades de java:** 233 archivos. 13215 líneas de código
- **Archivos XML:** 534 archivos. 86261 líneas de código.
- **Archivos Librerías JAR:** 534 archivos.

La herramienta se ejecutó con los parámetros que se observan en el bloque de código 5.2. Se proporcionó la ruta absoluta donde se encuentra el código fuente que se va a analizar, se especificó que se desea hacer un análisis completo de todas las secciones, y por último, se pidió buscar palabras clave del motor de base de datos Oracle.

```
1  # Comando de ejecución de programa.
2  rramirezespejo-macbookpro2:sap_commerce_analyzer rramirezespejo$ ./gradlew run
3  Starting a Gradle Daemon (subsequent builds will be faster)
4
5  > Task :app:run
6  [main] INFO co.edu.unal.utilities.ConsoleUtils -
       ----------------------------------------------
7  [main] INFO co.edu.unal.utilities.ConsoleUtils - - Welcome to SAP Commerce
       migration analyzer. -
8  [main] INFO co.edu.unal.utilities.ConsoleUtils -
       ----------------------------------------------
9  [main] INFO co.edu.unal.utilities.ConsoleUtils -
       ------------------------------------------------------------------
10 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Please enter the following
       details to proceed with the analysis -
11 [main] INFO co.edu.unal.utilities.ConsoleUtils -
       ------------------------------------------------------------------
12 [main] INFO co.edu.unal.utilities.InitUtils - Please enter the absolute path
       of the hybris dir to analyze:
13 # Ingresa directorio donde se encuentra el código fuente
14 /Users/rramirezespejo/dev_home/tiendaretailb2c
15 [main] INFO co.edu.unal.utilities.ConsoleUtils - Do you want to do a complete
       analysis?
16 (yes). no.
17 # No se ingresa ningún valor para usar (yes) por defecto.
18
```

```
19 [main] INFO co.edu.unal.utilities.InitUtils - ...Executing full analysis...
20 [main] INFO co.edu.unal.utilities.ConsoleUtils -
     --------------------------------------------------
21 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Which database keywords do
     you want to detect. -
22 [main] INFO co.edu.unal.utilities.ConsoleUtils -
     --------------------------------------------------
23 [main] INFO co.edu.unal.utilities.ConsoleUtils -
24 1. postgresql.
25 2. oracle.
26 3. mysql.
27 4. sqlserver.
28 5. saphana.
29
30 # Se selecciona la base de datos Oracle.
31 2
32 ...
33 ...
34 ...
35 [main] INFO co.edu.unal.utilities.ConsoleUtils -
     ------------------------------------------------------------
36 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Analysis ended. Results of
     the analysis were written in: -
37 [main] INFO co.edu.unal.utilities.ConsoleUtils -
     ------------------------------------------------------------
38 # Ruta en el sistema de archivos donde se encuentran los reportes con los
     resultados.
39 [main] INFO co.edu.unal.app.App - /Users/rramirezespejo/reports/analysis-
     report-20232026152021/
40
41 BUILD SUCCESSFUL in 1m 41s
42 28 actionable tasks: 1 executed, 27 up-to-date
```

Bloque de código 5.2: Parámetros de entrada evaluación tienda macromercado.

La evaluación realizada mediante la herramienta de análisis de código fuente en relación a la imple-

mentación del macrocomercio latinoamericano de SAP Commerce ha proporcionado los siguientes resultados (ver resumen en Tabla **5-2**).

Tabla **5-2**.: Resumen de resultados del estudio de caso 2: macromercado en línea.

Paquete	Archivo	Prioridad	Línea	Descripción	Conjunto de Reglas	Regla
Extensiones obsoletas						
	/b2cretailstore/bin /custom/b2cretail storecockpits /extensioninfo.xml	1	18	A extension info file is referencing a deprecated SAP Commerce extension	Deprecated extensions in ExtensionInfoFile	Deprecated Extension In Extension Info
	/b2cretailstore/build /config-dev /localextensions.xml	1	36	A local extensions file is referencing a deprecated SAP Commerce extension.	Deprecated extensions in LocalExtensions	Deprecated Extension In Local Extensions
Incompatibilidades de SAP Commerce						
	/b2cretailstore/build /config-dev /local.properties	1	8	Properties file contains a managed property 'db.username'.	Commerce Incompatibilities	Managed Properties Validation
	/b2cretailstore/build /config-dev /local.properties	2	8	Property 'db.username' contains the key '.username' which seems to contain sensitive information	Commerce Incompatibilities	Validate Sensitive Key
	/b2cretailstore/build /config-integ /local.properties	2	75	Properties file contains a duplicated property b2cretailstore .batch.export .newsletter .header	Commerce Incompatibilities	Validate Duplicates in file
Librerías de terceros						
	/b2cretailstore/bin /custom/partnerci /lib/jacocoant.jar	5		Jar with filename jacocoant not found in maven central.	Third party libraries	Jar Version Not Found In Maven Central

	/b2cretailstore/bin /custom /botonpagointegration /lib /powermock-api -support 1.6.1.jar	2		Newer jar version 2.0.9 found in maven central for jar powermock- api-support-1.6.1. Current version: 1.6.1	Third party libraries	Newer Jar Version In Maven Central
	/b2cretailstore/bin /custom/partnerci /lib/sonarqube-ant -task-2.5.jar	5		Jar with name sonarqube-ant-task-2.5 couldn't find the version in its manifest.	Third party libraries	Jar Version Not In Jar Manifest
Palabras clave de base de datos						
com. b2cretailstore .core.province .dao.impl	/b2cretailstore/bin /custom/b2cretail storecore/src/com /b2cretailstore /core /province /dao/impl /Defaultb2cretails toreProvinceDao.java	1	34	String literal "SELECT {p:PK} FROM { b2cretailstoreProvince As p} WHERE {p:provinceCode} = CAST (?provinceCode AS INTEGER)" contains the CONVERSION function 'CAST' which belongs to the database engine ORACLE.	Database keyword rule	Database Keyword Rule
com. b2cretailstore .core.province .dao.impl	/b2cretailstore/bin /custom/b2cretail storecore/src/com /b2cretailstore /core/province /dao/impl /Defaultb2cretail storeProvinceDao.java	1	52	String literal "SELECT {c:PK} FROM {b2cretailstoreCity AS c} WHERE {c:cityProvinceCode} = CAST (?provinceCode AS INTEGER) ORDER BY {c:cityName}" contains the CONVERSION function 'CAST' which belongs to the database engine ORACLE.	Database keyword rule	Database Keyword Rule

com. b2cretailstore .core.user .dao.impl	/b2cretailstore/bin /custom/b2cretail storecore/src/com /b2cretailstore /core/user /dao/impl /Defaultb2cretail storeUserDao.java	1	29	String literal "SELECT {c:pk} FROM {Customer AS c} WHERE to_char({c:modifiedTime}, 'YYYY/MM/DD HH:MI') > to_char({c: c4cIntegrationDate}, 'YYYY/MM/DD HH:MI') AND {c:uid} != 'anonymous' " contains the CONVERSION function 'TO_CHAR' which belongs to the database engine ORACLE	Database keyword rule	Database Keyword Rule.
Análisis de java						
com.partner .integration. botonpago .model.process	/b2cretailstore/bin /custom/botonpago integration /src/com/partner /integration/botonpago /model/process /RequestPayment Transaction.java	3	16	Avoid unnecessary constructors - the compiler will generate these for you.	Code Style	Unnecessary Constructor
com.partner .integration .botonpago .services	/b2cretailstore/bin /custom/botonpago integration /src/com/partner /integration/botonpago /services /BotonPago Service.java	1	58	Final parameter in abstract method	Code Style	Final Parameter In Abstract Method

com.partner .integration .botonpago .util	/b2cretailstore/bin /custom/boton-pago integration /src/com/partner /integration /botonpago /util /JaxbWrapper.java	2	56	Logger calls should be surrounded by log level guards.	Best Practices	Guard Log Statement
Análisis de javascript						
	/b2cretailstore/bin /custom /b2cretail store storefront /Gruntfile.js	1	63	Avoid trailing commas in object or array literals	Error Prone	Avoid Trailing Comma
	/b2cretailstore/bin /custom /b2cretail store storefront /web/webroot /WEB-INF /_ui-src /responsive /lib /bootstrap-3.3.7 /js /dropdown.js	2	94	A function should not mix return statements with and without a result.	Best Practices	Consistent Return
	/b2cretailstore/bin /custom /b2cretail store storefront /web/webroot /WEB-INF /_ui-src /responsive /lib /bootstrap-3.3.7 /js /dropdown.js	3	107	Use ===/!== to compare with true/false or Numbers	Error Prone	Equal Comparison

Fuente: Elaboración propia

A partir de los hallazgos encontrados, vale la pena mencionar los siguientes:

- **Total de problemas identificados:** La evaluación reveló un total de 7388 problemas encontrados en el código fuente de la implementación personalizada de “TiendaRetailB2C”.

- **Tipos y severidad de los problemas:** Los 7388 problemas encontrados se discriminan de la siguiente forma:
  1. **Extensiones obsoletas:** 31 referencias a extensiones obsoletas de prioridad Alta.
  2. **Incompatibilidades de SAP Commerce:** 233 hallazgos - 77 prioridad alta y 156 prioridad media-alta.
  3. **Librerías de terceros:** 24 hallazgos - 11 prioridad media-alta y 13 prioridad baja.
  4. **Palabras clave de base de datos:** 17 hallazgos de prioridad alta.
  5. **Análisis de Java:** 1495 hallazgos - 107 prioridad alta, 307 prioridad media-alta, 760 prioridad media y 321 prioridad media-baja.
  6. **Análisis de Javascript:** 5612 hallazgos - 531 prioridad alta, 4738 prioridad media-alta y 342 prioridad media.

En primer lugar, es importante destacar que se han identificado 31 referencias a extensiones obsoletas en el código fuente. Este hallazgo se debe a que el código fuente fue desarrollado para la versión 6.3.0 de SAP Commerce, la cual no está respaldada por la arquitectura de nube pública según la documentación correspondiente[1].

La información anterior sugiere que el proceso de adaptación de esta implementación puede requerir un esfuerzo considerable. Por lo tanto, los consultores de SAP Commerce encargados de evaluar la viabilidad de esta migración deben realizar un análisis exhaustivo para determinar si resulta más conveniente utilizar una versión actualizada y volver a implementar las personalizaciones, en lugar de intentar adaptar el código fuente de una versión desarrollada hace seis años. Dicho análisis detallado permitirá evaluar la complejidad y los posibles desafíos involucrados en cada enfoque, considerando factores como la estabilidad, la eficiencia y el mantenimiento futuro del sistema.

Por otro lado, se ha detectado la presencia de 17 hallazgos de palabras clave relacionadas con las funciones del motor de base de datos Oracle, lo cual señala una dependencia significativa de esta tecnología de base de datos. Si bien sería válido considerar reemplazar estas palabras clave con funciones compatibles con Azure SQL, no se recomienda esta acción debido a que SAP está colaborando

[1] https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/12be4ac419604b01aabb1adeb2c4c8a2/1c6c687ad0ed4964bb43d409818d23a2.html

con varios proveedores de servicios en la nube para migrar sus soluciones. Esto implica que la portabilidad del código se vería comprometida al intentar implementar la solución en proveedores de nube distintos a Azure.

En relación con las librerías de terceros, se han identificado un total de 24 hallazgos. De estos, 11 librerías requieren una actualización, mientras que para 13 de ellas no se encontró la versión declarada en el manifiesto de la librería. SAP Commerce enfatiza la importancia de mantener actualizadas las librerías de terceros, preferiblemente a su última versión disponible. Esta práctica contribuye a evitar problemas de seguridad y permite aprovechar las mejoras implementadas por los desarrolladores en cada versión. Asimismo, mantener las librerías actualizadas brinda beneficios adicionales en términos de funcionalidad y rendimiento del sistema.

Los resultados obtenidos del análisis de incompatibilidades en SAP Commerce revelaron la existencia de un considerable número de configuraciones de infraestructura manejadas por SAP en la arquitectura en la nube. Estas configuraciones no representan un esfuerzo significativo, ya que simplemente deben ser eliminadas. Por otro lado, se identificó la presencia de numerosas claves, usuarios, llaves, IDs y otras propiedades sensibles declaradas en los archivos de propiedades de SAP Commerce. Estas configuraciones probablemente se utilizan para establecer el intercambio de claves para integraciones. Sin embargo, es considerada una mala práctica tener estas credenciales en blanco en el código fuente, ya que expone información sensible de forma innecesaria.

Por último, se han identificado numerosos problemas relacionados con el análisis de buenas prácticas en los lenguajes de propósito general Java y JavaScript. Específicamente, se encontraron 107 hallazgos de prioridad alta en Java, mientras que en JavaScript se registraron 531. Estos resultados indican que los desarrolladores no han establecido una cultura de análisis del código fuente durante el proceso de desarrollo, lo que ha generado una deuda técnica significativa que requiere ser reducida.

En el caso particular de JavaScript, muchos de estos problemas están asociados a librerías externas como jQuery. Esto plantea la necesidad de reevaluar su uso dentro de SAP Commerce, y se recomienda considerar la adopción de frameworks web más modernos, como Angular, Vue.js, React, entre otros. Estos frameworks ofrecen mejoras en cuanto a estructura, rendimiento y mantenibilidad del código, lo que puede contribuir a superar los problemas identificados y fomentar una base de código más robusta y actualizada.

En conclusión, los hallazgos presentados en esta evaluación indican de manera general que adaptar la implementación actual a la arquitectura en la nube representa un esfuerzo considerablemente mayor que reimplementar las características personalizadas en una versión más reciente de SAP Commerce. Incluso el simple hecho de eliminar las referencias a las extensiones obsoletas puede requerir un esfuerzo considerable, superando en muchos casos el esfuerzo necesario para llevar a cabo una reimplementación completa de la solución desde cero.

5.4. Estudio de caso 3: Tienda en línea entre comercios de artículos electrónicos

La herramienta de análisis automático de código fuente fue evaluada utilizando un último estudio de caso que involucra otra implementación real de un comercio en línea en SAP Commerce. En esta ocasión, el código fuente evaluado corresponde a una tienda en línea basada en el modelo de comercio a comercio, es decir, un proveedor de artículos electrónicos de diversas categorías que suministra estos productos a otros negocios para su línea de producción. Este negocio tiene presencia en Estados Unidos, Canadá y Europa Occidental.

Al igual que en el estudio de caso anterior, se obtuvo autorización verbal para llevar a cabo las pruebas en el código fuente de forma verbal, con la condición de mantener el anonimato de este negocio. Por lo tanto, con el objetivo de preservar la confidencialidad, se hará referencia a esta empresa con el pseudónimo de "B2BTechRetailStore". En esta implementación en particular, se utilizó SAP Commerce con el propósito de aprovechar su módulo de administración y gestión de productos, inventarios, pedidos y otras características relacionadas con el proceso de venta en línea entre empresas. El código fuente evaluado posee las siguientes características:

- **Versión de Java:** Open JDK 11.
- **Versión de SAP Commerce:** 1808.
- **Motor de base de datos:** Oracle 11g.
- **Total de archivos:** 2650 archivos. 315170 líneas de código.

- **Archivos de Java:** 1693 archivos. 183694 líneas de código.
- **Archivos de Javascript:** 4 archivos. 298 líneas de código.
- **Archivos de propiedades de java:** 373 archivos. 14427 líneas de código
- **Archivos XML:** 389 archivos. 143014 líneas de código.
- **Archivos Librerías JAR:** 36 archivos.

La herramienta se ejecutó con los parámetros que se observan en el bloque de código 5.3. Se proporcionó la ruta absoluta donde se encuentra el código fuente que se va a analizar, se especificó que se desea hacer un análisis completo de todas las secciones, y por último, se pidió buscar palabras clave del motor de base de datos Oracle.

```
1  # Comando de ejecución de programa.
2  rramirezespejo-macbookpro2:sap_commerce_analyzer rramirezespejo$ ./gradlew run
3  Starting a Gradle Daemon (subsequent builds will be faster)
4  ...
5  ...
6  ...
7  [main] INFO co.edu.unal.utilities.ConsoleUtils -
       -----------------------------------------------------------------
8  [main] INFO co.edu.unal.utilities.InitUtils - Please enter the absolute path
       of the hybris dir to analyze:
9  # Se ingresa ruta absoluta del sistema de archivos donde se encuentra el
       código fuente
10 /Users/rramirezespejo/dev_home/b2btechretailstore/hybris/bin/custom
11 [main] INFO co.edu.unal.utilities.ConsoleUtils - Do you want to do a complete
       analysis?
12 (yes). no.
13 # Se deja vacío para utilizar (yes) por defecto.
14 
15 [main] INFO co.edu.unal.utilities.ConsoleUtils -
       --------------------------------------------------
16 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Which database keywords do
       you want to detect. -
```

```
17 [main] INFO co.edu.unal.utilities.ConsoleUtils -
   --------------------------------------------------
18 [main] INFO co.edu.unal.utilities.ConsoleUtils -
19 1. postgresql.
20 2. oracle.
21 3. mysql.
22 4. sqlserver.
23 5. saphana.
24 # Se indica el motor de base de datos oracle.
25 
26 2
27 ...
28 ...
29 ...
30 [main] INFO co.edu.unal.utilities.ConsoleUtils -
   ------------------------------------------------------------
31 [main] INFO co.edu.unal.utilities.ConsoleUtils - - Analysis ended. Results of
   the analysis were written in: -
32 [main] INFO co.edu.unal.utilities.ConsoleUtils -
   ------------------------------------------------------------
33 # Ruta absoluta del sistema de archivos donde se encuentran los resutlados.
34 [main] INFO co.edu.unal.app.App - /Users/rramirezespejo/reports/analysis-
   report-20234227184216/
35 
36 BUILD SUCCESSFUL in 43s
37 28 actionable tasks: 1 executed, 27 up-to-date
```

Bloque de código 5.3: Parámetros de entrada evaluación Tienda negocio a negocio de artículos electrónicos.

La evaluación realizada mediante la herramienta de análisis de código fuente, en relación, a la implementación del distribuidor de artículos electrónicos norteamericano de SAP Commerce, ha proporcionado los siguientes resultados (ver resumen en Tabla **5-3**):

Tabla **5-3**.: Resumen de resultados del estudio de caso 3: tienda en línea entre comercios de artículos electrónicos.

Paquete	Archivo	Prioridad	Línea	Descripción	Conjunto de Reglas	Regla
Extensiones obsoletas						
	/b2btechretailstore /repository /hybris /config /localextensions.xml	1	10	A local extensions file is referencing a deprecated SAP Commerce extension.	Deprecated extensions in LocalExtensions	Deprecated Extension In Local Extensions
	/b2btechretailstore /repository /hybris /config /localextensions.xml	1	13	A local extensions file is referencing a deprecated SAP Commerce extension.	Deprecated extensions in LocalExtensions	Deprecated Extension In Local Extensions
	/b2btechretailstore /repository /hybris/env/common /localextensions.xml	1	10	A local extensions file is referencing a deprecated SAP Commerce extension.	Deprecated extensions in LocalExtensions	Deprecated Extension In Local Extensions
	/b2btechretailstore /repository /hybris/env/common /localextensions.xml	1	13	A local extensions file is referencing a deprecated SAP Commerce extension.	Deprecated extensions in LocalExtensions	Deprecated Extension In Local Extensions
Incomptaibilidades de SAP Commerce						
	/b2btechretailstore /repository/hybris /bin/custom /b2btechretailstore webservices /web/webroot/WEB-INF /messages /messages_ru.properties	2	35	Property 'field.password .min.six .characters' contains the key '.password' which seems to contains sensitive information	Commerce Incompatibilities	Validate Sensitive Key
	/b2btechretailstore /repository/hybris /env/dr /db.properties	1	4	Properties file contains a managed property 'db.url'.	Commerce Incompatibilities	Managed Properties Validation

	/b2btechretailstore /repository/hybris /env/qa3 /local.properties	1	10	Properties file contains a managed property 'regioncache .entityregion .size'.	Commerce Incompatibilities	Managed Properties Validation
Librerías de terceros						
	/b2btechretailstore /repository/hybris /bin/custom /b2btechretail storedealreg/lib /gson-2.8.2 -SNAPSHOT.jar	5		Jar with filename gson-2.8.2-SNAPSHOT not found in maven central.	Third party libraries	Jar Version Not Found In Maven Central
	/b2btechretailstore /repository/hybris /bin/custom /b2btechretail storewebservices /web/webroot /WEB-INF/lib /jersey-test -framework-core-1.13.jar	2		Newer jar version 3.1.2 found in maven central for jar jersey-test-framework-core-1.13. Current version: 1.13	Third party libraries	Newer Jar Version In Maven Central
	/b2btechretailstore /repository /hybris/-bin /custom/ruleengine /b2btechretail storeruleengine commonsbackoffice /resources /backoffice /b2btechretailstore ruleengine commons-backoffice_bof.jar	5		Jar with name b2btech retailstore ruleengine commonsbackoffice_bof couldn't find the version in its manifest.	Third party libraries	Jar Version Not In Jar Manifest
Palabras clave de base de datos						

com. b2btech retails-tore .hy-bris.cockpits .importcock-pit .dao.impl	/b2btechretailstore /repository/hy-bris /bin/custom /b2btechretailstore /b2btech retails-torecore /src/-com /b2btech retailstore /hybris /cockpits/import-cockpit /dao/impl /Defaultb2btech retailstoreImport CockpitCronJob-Dao.java	1	57	String literal "WHERE LO-WER({ p." contains the STRING function 'LOWER' which belongs to the databa-se engine ORACLE.	Database key-word rule	Database Key-word Rule
com.b2btech retailstore .hybris.core .daos.impl	/b2btechretailstore /repository/hy-bris /bin/custom /b2btechretailstore /b2btechretailstorecore /src/com /b2btechretailstore /hybris/core/-daos /impl /De-faultb2btech re-tailstore PriceRow-Dao.java	1	118	String literal "AND TO_DATE(%s, ' %s') $>=$?currentDate" contains the CONVERSION function 'TO_DATE' which belongs to the database engine ORACLE.	Database key-word rule	Database Key-word Rule

com .b2btech retailstore .hybris.core .sugges- tion.dao .impl	/b2btechretailstore /repository/hy- bris /bin/custom /b2btechretailstore /b2btech retails- torecore /src/com /b2btechretailstore /hybris/core /sug- gestion/dao /impl /DefaultSimple Sug- gestionDao.java	1	53	String literal “SE- LECT DISTINCT {p.PK}, COUNT({p.PK}) AS NUM” contains the NUMERIC function ’COUNT’ which belongs to the database engine ’ORACLE’.	Database key- word rule	Database Key- word Rule
Análisis de Java						
com .b2btech retailstore .hybris.core .attributes	/b2btechretailstore /repository/hy- bris /bin/custom /b2btechretailstore /b2btechretail sto- recore /src/com /b2btechretailstore /hybris/core /attri- butes /CNETMedia URLAttrHandler.java	1	20	The method parameter name ’URL’ doesn’t match ’[a-z][a- zA-Z0-9]*’	Code Style	Formal Para- meter Naming Conventions

com .b2btech retails-tore .hy-bris.aspects	/b2btechretailstore /repository/hy-bris /bin/custom /b2btechretailstore /b2btechretail sto-recore /src/com /b2btechretailstore /hybris/aspects /Get-MethodAspect.java	2	30	Use of modifier volatile is not recommended.	Multithreading	Avoid Using Volatile
com .b2btech retails-tore .hy-bris.cockpits .importcock-pit .servi-ces.media .impl	/b2btechretailstore /repository/hy-bris /bin/custom /b2btechretailstore /b2btechretail sto-recockpits /src/com /b2btechretailstore /hybris/cockpits /im-portcockpit/services /media/impl /De-faultb2btech retails-tore ImportCockpit MediaService.java	3	156	Do not use ‘new Charac-ter(...)‘, prefer ‘Charac-ter.valueOf(...)‘	Best Practices	Primitive Wrapper Ins-tantiation

com .b2btech retails- tore .hy- bris.cockpits .importcock- pit .servi- ces.impex .genera- tor.operations .impl	/b2btechretailstore /repository/hy- bris /bin/custom /b2btechretailstore /b2btechretail sto- recockpits /src/com /b2btechretailstore /hybris/cockpits /importcockpit /services/impex /ge- nerator/operations /impl /b2btechretail storeImpEx Transfor- mation Service.java	4	3	Unused import 'com.b2btechretailstore .hybris.core .services .Ven- dorService'	Code Style	Unnecessary Import
Análisis de Javascript						
	/b2btechretailstore /repository/hybris /env/common /cus- tomize/platform /ext/hac /web/- webroot /static/js /home/index.js	1	103	Avoid using global variables	Best Practices	Global Varia- ble
	/b2btechretailstore /repository/hybris /env/common /cus- tomize/platform /ext/hac/web /we- broot/static /js/home /index.js	1	156	The for-in loop variable 'pos' should be explicitly scoped with var to avoid pollution.	Best Practices	Scope For In Variable

	/b2btechretailstore /repository/hybris /env/common /customize/platform /ext/hac/web /webroot/static /js/home /index.js	2	269	The numeric literal '172800000' will have at different value at runtime.	Error Prone	Innaccurate Numeric Literal

Fuente: Elaboración propia

- **Total de problemas identificados:** La evaluación reveló un total de 2841 problemas encontrados en el código fuente de la implementación personalizada de "B2BTechRetailStore".
- **Tipos y severidad de los problemas:** Los 2841 problemas encontrados se discriminan de la siguiente forma:
  1. **Extensiones obsoletas:** 4 referencias a 2 extensiones obsoletas de prioridad Alta.
  2. **Incompatibilidades de SAP Commerce:** 286 hallazgos - 92 prioridad alta y 194 prioridad media-alta.
  3. **Librerías de terceros:** 23 hallazgos - 11 prioridad media-alta y 12 prioridad baja.
  4. **Palabras clave de base de datos:** 51 hallazgos de prioridad alta.
  5. **Análisis de Java:** 2041 hallazgos - 193 prioridad alta, 519 prioridad media-alta, 679 prioridad media y 650 prioridad media-baja.
  6. **Análisis de Javascript:** 436 hallazgos - 47 prioridad alta, 389 prioridad media-alta y 342 prioridad media.

Los resultados de este caso revelan la existencia de dos puntos principales que obstaculizan la migración. En primer lugar, es necesario eliminar la dependencia de las dos extensiones obsoletas. En segundo lugar, se debe abordar la eliminación de las palabras clave relacionadas con la base de datos. Se encontraron 51 hallazgos que representan un bloqueo para la migración exitosa. Para resolver estos 51 hallazgos, se requiere eliminar las funciones específicas de la base de datos Oracle y reem-

plazarlas con procesamiento utilizando Java, o bien replantear completamente el proceso de consulta de la información de la base de datos.

Estos dos puntos de bloqueo constituyen desafíos significativos que deben abordarse para lograr una migración exitosa. La eliminación de las extensiones obsoletas y la adaptación de las funciones de la base de datos son aspectos críticos para garantizar la compatibilidad y el funcionamiento adecuado de la implementación en un entorno de nube. Se requiere un enfoque cuidadoso y un análisis detallado para implementar las soluciones adecuadas y asegurar una migración exitosa y sin problemas.

Una vez que se hayan abordado los puntos mencionados anteriormente, es necesario resolver las incompatibilidades de infraestructura relacionadas con las propiedades administradas. Afortunadamente, este proceso no representa un esfuerzo significativo, ya que simplemente implica la eliminación de estas propiedades de los archivos correspondientes.

Al remover las propiedades manejadas que generan incompatibilidades de infraestructura, se logrará un mayor alineamiento con la arquitectura en la nube deseada. Esta tarea se enfoca principalmente en asegurar que las configuraciones y propiedades se ajusten adecuadamente a los estándares y requisitos de la infraestructura en la nube. Mediante la eliminación de estas propiedades, se evitan posibles conflictos y se facilita la migración hacia la arquitectura en la nube de manera más fluida y eficiente.

Por último, en lo que respecta a las librerías de terceros, es necesario realizar la actualización de las 11 librerías identificadas. Para el lenguaje de programación Java, también se debe abordar la deuda técnica relacionada con los hallazgos de prioridad alta y media-alta. En el caso de JavaScript, no se requiere realizar un esfuerzo significativo, dado que la implementación de este cliente no implica una interfaz web. En su lugar, se utilizan los módulos de administración de datos para intercambiar información con los clientes mediante integraciones.

La actualización de las librerías de terceros es fundamental para mantener la seguridad y la eficiencia del sistema, ya que las nuevas versiones suelen incluir mejoras, correcciones de errores y actualizaciones de seguridad importantes. Por otro lado, abordar la deuda técnica en el código Java ayudará a mejorar la calidad y mantenibilidad del código, reduciendo la posibilidad de errores y facilitando futuras actualizaciones.

En conclusión, a diferencia del estudio de caso anterior, a nivel general es factible considerar una adaptación del código fuente para abordar y eliminar los hallazgos que representan bloqueos en la

migración. Una vez resueltos estos obstáculos, se recomienda elaborar un plan para reducir la deuda técnica, centrándose especialmente en el código fuente de Java. Al seguir este enfoque, se podrá lograr una implementación más robusta, escalable y mantenible, lo que permitirá aprovechar al máximo los beneficios de la migración a la arquitectura en la nube y asegurar el éxito a largo plazo del proyecto.

6. Conclusiones y trabajo futuro

El análisis automático de código fuente es una herramienta poderosa que puede ayudar a los consultores de SAP Commerce a ahorrar tiempo y esfuerzo. Mediante la estrategia propuesta y la herramienta implementada se puede ejecutar el análisis de forma automática, lo que libera a los consultores para que se concentren en otras tareas. Además, la herramienta puede identificar problemas potenciales en el código fuente, lo que puede ayudar a los clientes y desarrolladores a evitar errores.

La herramienta de análisis automático de código fuente puede ayudar a acelerar las migraciones de SAP Commerce al detectar algunos errores comunes fáciles de corregir. Al identificar estos errores en una etapa temprana, la herramienta puede ayudar a evitar retrasos en el proyecto. Además, puede ayudar a asegurar que la migración sea exitosa y que el sistema siga las buenas prácticas recomendadas por SAP.

Uno de los objetivos principales de la evaluación realizada en tres estudios de caso distintos es proponer el uso constante de la herramienta de análisis automático de código fuente durante las evaluaciones realizadas por los consultores de SAP, con el fin de determinar el esfuerzo y enfoque requeridos para abordar una migración a la nube. A partir de los casos de estudio de SAP Commerce 123 y la tienda distribuidora de artículos electrónicos, se pudo determinar que se necesitan adaptaciones y la eliminación de hallazgos bloqueantes para lograr una migración exitosa. Sin embargo, en el caso del macromercado, se identificó una fuerte dependencia en extensiones obsoletas, sumado a otros factores como la antigüedad del código fuente, lo cual proporciona elementos que sugieren a los consultores recomendar un enfoque de migración basado en la reimplementación en lugar de la adaptación del código existente.

El uso constante de la herramienta de análisis automático de código fuente en las evaluaciones permite a los consultores obtener una visión clara de los desafíos y obstáculos que se presentan en la

migración a la nube. Esto proporciona una base sólida para tomar decisiones informadas y ofrecer recomendaciones adecuadas a los clientes. Al identificar de manera precisa los hallazgos bloqueantes y evaluar la viabilidad de la adaptación o reimplementación, los consultores pueden diseñar estrategias efectivas y eficientes para llevar a cabo la migración exitosa de SAP Commerce.

Por otra parte, el análisis automático de código fuente puede ayudar a asegurar la adopción de buenas prácticas para infraestructuras basadas en la nube y para lenguajes de propósito general como Java y JavaScript al identificar y corregir violaciones de las mejores prácticas. Al identificar estas violaciones, la herramienta puede ayudar a garantizar que la infraestructura basada en la nube y los sistemas que se implementen en ella sean seguros, eficientes y escalables.

Es importante destacar que los resultados proporcionados por esta herramienta contribuyen a la identificación de errores comunes y a la promoción de buenas prácticas. Sin embargo, es fundamental tener en consideración que la herramienta no es completamente infalible, por lo que es necesario continuar expandiendo este trabajo para incorporar características y funcionalidades que no se han abordado en este trabajo final de maestría.

El trabajo de análisis automático de código fuente puede extenderse con la ayuda de la analítica de software no sólo para detectar errores, sino también para corregirlos, o implementar estrategias preventivas para las migraciones de SAP Commerce a la infraestructura de la nube. Por ejemplo, la analítica de software puede utilizarse para: proponer soluciones a los errores identificados, implementar las soluciones propuestas, monitorizar el rendimiento del sistema después de la migración para asegurarse de que no hay problemas.

Adicionalmente, la analítica de software puede utilizarse para implementar estrategias preventivas para evitar que se produzcan errores en el código fuente. Por ejemplo también es posible: identificar patrones en el código fuente que pueden indicar problemas potenciales; generar informes sobre el estado del código fuente, que pueden utilizarse para identificar áreas que necesitan atención; comparar diferentes versiones del código fuente, lo que puede utilizarse para identificar cambios que pueden haber afectado al rendimiento, la seguridad o la mantenibilidad.

El trabajo presentado en su estado actual puede extenderse mediante el uso de herramientas de integración continua como Git, GitHub Actions, Jenkins. Estas herramientas permiten automatizar el proceso de desarrollo, desde la compilación del código hasta su prueba y despliegue. En general, el uso

de herramientas de integración continua puede ayudar a mejorar la calidad, la seguridad y el tiempo de comercialización del software. Esto puede ser beneficioso para las empresas de todos los tamaños. Por ejemplo, Git puede ser integrado para descargar el código de forma automática sin intervención humana. Esto puede liberar a los desarrolladores para que se centren en otras tareas, como escribir código nuevo o mejorar el código existente. También se puede crear un plugin de GitHub Actions para analizar el código directamente en GitHub. Esto puede ayudar a identificar errores y vulnerabilidades en el código, lo que puede mejorar la seguridad del software. Por último, el código también puede ser agregado como un plugin para el sistema de Jenkins. Esto puede ayudar a automatizar el proceso de despliegue del sistema, lo que puede acelerar el tiempo de comercialización del software.

Referencias

Akiyama, F. (1971). An example of software system debugging. In *IFIP Congress*.

Anderson, W. (1987). Software validation techniques. *Therapeutic Innovation & Regulatory Science*, 21(4):461–469.

Ann, G., Pbell, C., and Sa, S. C. (2021). A new way of measuring understandability cognitive complexity cognitive complexity-a new way of measuring understandability. *Sonarsource*.

Antal, G., Szarka, A., and Hegedűs, P. (2018). *A hands-on openstack code refactoring experience report*, volume 10964 LNCS. Springer Science+Business Media.

Anwar, H. and Pfahl, D. (2017). Towards greener software engineering using software analytics: A systematic mapping. In *Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017*, pages 157–166.

Apache (2023). Apache license. `https://www.apache.org/licenses/`.

Ashfaq, Q., Khan, R., and Farooq, S. (2019). A comparative analysis of static code analysis tools that check java code adherence to java coding standards. *2019 2nd International Conference on Communication, Computing and Digital Systems, C-CODE 2019*, pages 98–103.

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., and Penix, J. (2008). Using static analysis to find bugs. *IEEE Software*, 25(5):22–29.

Binkley, D. (2007). Source code analysis: A road map. In *Source code analysis: A road map*, pages 104–119.

Blanchard, A., Loulergue, F., and Kosmatov, N. (2019). Towards full proof automation in Frama-C using auto-active verification. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11460 LNCS, pages 88–105.

Blumenthal, J. (2022). Thinking like a lawyer: Why you or your it team needs to keep your software systems up-to-date. *SIGCAS Comput. Soc.*, 50(3):10.

Bryman, A. (2016). *Social research methods*. Oxford university press.

Buse, R. P. and Zimmermann, T. (2012). Information needs for software development analytics. In *Proceedings - International Conference on Software Engineering*, pages 987–996.

Ceron, H. (2022). Cx works | sap commerce cloud architecture. https://www.sap.com/cxworks/article/2589633403/sap_commerce_cloud_architecture. [Online; accessed 6-February-2022].

Chen, Y., Santosa, A. E., Yi, A. M., Sharma, A., Sharma, A., and Lo, D. (2020). A machine learning approach for vulnerability curation. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pages 32–42.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-C: A software analysis perspective. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7504 LNCS, pages 233–247.

DeepCode, CodeScene, Semgrep, Codiga, and FFensive360 (2022). Best 104 java static analysis tools and linters. https://analysis-tools.dev/tag/java. [Online; accessed 27-November-2022].

Devanbu, P., Zimmermann, T., and Bird, C. (2016). Belief & evidence in empirical software engineering. In *Proceedings - International Conference on Software Engineering*, volume 14-22-May-, pages 108–119.

Golubev, Y., Eliseeva, M., JetBrains, N. P., and Bryksin, T. (2020). A study of potential code borrowing and license violations in java projects on github. *arxiv*.

GradleInc (2023a). Gradle license information. https://docs.gradle.org/current/userguide/licenses.html.

GradleInc (2023b). Organizing gradle projects. https://docs.gradle.org/current/userguide/organizing_gradle_projects.html#sec:build_sources.

GradleInc (2023c). Structuring and building a software component with gradle. https://docs.gradle.org/current/userguide/multi_project_builds.html.

Huijgens, H., Spadini, D., Stevens, D., Visser, N., and Van Deursen, A. (2018). Software analytics in continuous delivery: A case study on success factors. In *International Symposium on Empirical Software Engineering and Measurement*.

Lanza, M. and Marinescu, R. (2006). Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, pages 1–205.

Lathar, P., Shah, R., and G, S. K. (2017). Stacy-static code analysis for enhanced vulnerability detection under a creative commons attribution (cc-by) 4.0 license stacy-static code analysis for enhanced vulnerability detection. *Cogent Engineering*, 4:1335470.

Lieberherr, K. and Holland, I. (2023). Law of demeter: Principle of least knowledge. `https://www.ccs.neu.edu/home/lieber/LoD.html`. [Online; accessed 13-Februrary-2023].

MacWilliam, T. (2022). Cx works | measuring code quality with sonar. `https://www.sap.com/cxworks/article/2589634066/measuring_code_quality_with_sonar`. [Online; accessed 6-February-2022].

Mauerer, W. and Scherzinger, S. (2020). Educating future software architects in the art and science of analysing software data. In *CEUR Workshop Proceedings*, volume 2531, pages 56–60.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Menzies, T. (2019). Take Control : (On the Unreasonable Effectiveness of Software Analytics). In *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, pages 265–266.

Menzies, T. and Zimmermann, T. (2013). Software analytics: So what? *IEEE Software*, 30(4):31–37.

Menzies, T. and Zimmermann, T. (2018). Software Analytics: What's Next? *IEEE Software*, 35(5):64–70.

Ohlsson, N., Zhao, M., and Helander, M. (1998). Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 7(1):51–66.

OracleCorp (2023). Openjdk license. `https://openjdk.org/legal/gplv2+ce.html`.

Park, W. and Lee, I. (2023). Log4j vulnerability analysis and detection pattern production technology

based on snort rules. *Studies in Computational Intelligence*, 1075:165 – 174. Cited by: 0.

Pfeiffer, R.-H. (2021). Identifying critical projects via pagerank and truck factor. In *Identifying Critical Projects via PageRank and Truck Factor*.

PMD (2022). Ecmascript rules | pmd source code analyzer. https://pmd.github.io/latest/pmd_rules_ecmascript.html. [Online; accessed 28-November-2022].

PMD (2023). Pmd license. https://docs.pmd-code.org/latest/license.html.

Porter, J. D. (2023). Capitalization rules. https://wiki.c2.com/?CapitalizationRules. [Online; accessed 13-Februrary-2023].

Pruijt, L., Köppe, C., and Brinkkemper, S. (2013). Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support. In *IEEE International Conference on Software Maintenance, ICSM*, pages 220–229.

Raibulet, C. and Arcelli Fontana, F. (2018). Collaborative and teamwork software development in an undergraduate software engineering course. *Journal of Systems and Software*, 144:409–422.

Robles, G. and Gonzalez-Barahona, J. M. (2013). Mining student repositories to gain learning analytics. An experience report. In *IEEE Global Engineering Education Conference, EDUCON*, pages 1249–1254.

SAP (2020). E-commerce platform: Headless e-commerce in the cloud | sap. https://www.sap.com/products/crm/e-commerce-platforms.html?btp=6cbf1321-d49b-4fb2-bf3e-f38e1a4a0911.

SAP About (2022). About SAP Commerce - SAP Help Portal.

SAP Help (2022). Sap help | development best practices. https://help.sap.com/viewer/129a68efcdaf43dc94243b57f9aba5ad/2105/en-US/2978af7252c4492da6b8a5f1a89c1da5.html. [Online; accessed 6-February-2022].

SAP Local extensions (2023). The localextensions.xml file | sap help portal - sap help portal. https://help.sap.com/docs/SAP_COMMERCE/3fb5dcdfe37f40edbac7098ed40442c0/31bf348455034a2299d34626c928309e.html?locale=en-US.

SAP Managed Properties (2023). Managedproperties | sap help portal - sap help portal. https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/

1be46286b36a4aa48205be5a96240672/a30160b786b545959184898b51c737fa.html?locale=en-US.

SAP Sensitive configuration (2023). Uploading sensitive configuration data | sap help portal - sap help portal. https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/0fa6bcf4736c46f78c248512391eb467/557da644ed1949de9d40a3e4615d8b9b.html?locale=en-US.

Signoles, J. (2015). Software architecture of code analysis frameworks matters: The Frama-C example. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 187, pages 86–96.

SLF4J (2023). Licensing terms for slf4j. https://www.slf4j.org/license.html.

Soltanifar, B., Akbarinasaji, S., Caglayan, B., Bener, A., Filiz, A., and Kramer, B. (2016). Software analytics in practice: A defect prediction model using code smells. In *ACM International Conference Proceeding Series*, volume 11-13-July, pages 148–155.

SonarSource (2022). About sonarqube. https://www.sonarqube.org/about/. [Online; accessed 6-February-2022].

SonatypeInc (2023). The central repository documentation | rest api. https://central.sonatype.org/search/rest-api-guide/.

Spring (2020). Spring | why spring? https://spring.io/why-spring.

Wilkes, M. (1985). *Memoirs of a Computer Pioneer*. Massachusetts Institute of Technology, USA.

Zimmermann, T. (2015). Software Analytics for Digital Games. In *Proceedings - 4th International Workshop on Games and Software Engineering, GAS 2015*, pages 1–2.

A. Código ilustrativo de reglas de análisis de código fuente

Este anexo incluye los bloques de código de ejemplo correspondientes a las reglas definidas para la estrategia de análisis de código fuente en SAP Commerce, tal como se presenta en el Capítulo 3. Este anexo se divide en dos secciones principales:

Sección A.1: Ejemplos de buenas prácticas, estilo de código, diseño, documentación, propensión a errores, multihilo y seguridad para el lenguaje de propósito general Java. En esta sección, se presentan los ejemplos de código que ilustran las buenas prácticas recomendadas, los estándares de estilo de código, el diseño eficiente, la documentación adecuada, la identificación de posibles errores, la programación multihilo y las consideraciones de seguridad en el contexto de Java.

Sección A.2: Ejemplos de buenas prácticas, estilo de código y propensión a errores para el lenguaje de propósito general Javascript. En esta sección, se proporcionan ejemplos de código que ejemplifican las buenas prácticas, el estilo de código recomendado y la identificación de posibles errores en el contexto de Javascript.

Estos bloques de código de ejemplo son una herramienta útil para comprender las reglas definidas en la estrategia de análisis de código fuente. Al estudiar y analizar estos ejemplos, es posible familiarizarse con las prácticas recomendadas y mejorar la calidad del código en sus implementaciones de SAP Commerce.

A.1. Bloques de código reglas de Java

A.1.1. Ejemplos de buenas prácticas de Java

Volver a descripción de la regla.

```
1 public abstract class Foo {
2   void int method1() { ... }
3   void int method2() { ... }
4   // Considere usar metodos abstractos o remueva el modificador abstracto
5   // y agregue metodos de tipo protected
6 }
```

Bloque de código A.1: Evite el uso de clases abstractas sin métodos abstractos.

Volver a descripción de la regla.

```
1 public class Outer {
2  void method(){
3   Inner ic = new Inner(); // Causa la generacion de un descriptor de accesos
      de clase.
4  }
5  public class Inner {
6   private Inner(){}
7  }
8 }
```

Bloque de código A.2: Regla Java: Generación de descriptor de acceso de clase.

Volver a descripción de la regla.

```
1 public class OuterClass {
2     private int counter;
3     /* modificador de tipo paquete */ int id;
4     public class InnerClass {
5         InnerClass() {
6             OuterClass.this.counter++; // Incorrecto: Método de accesos para
                  la variable counter se generará.
7         }
8         public int getOuterClassId() {
```

```
9             return OuterClass.this.id; // el atributo id es de tipo package,
                por lo tanto ningún método de acceso es necesario.
10        }
11    }
12 }
```

Bloque de código A.3: Regla Java: Generación de descriptor de acceso de método.

Volver a descripción de la regla.

```
1 public class Foo {
2     private String [] x;
3         public void foo (String [] param) {
4         // Evite la asignación directa, es mejor crear una nueva copia del
              arreglo.
5         this.x=param;
6     }
7 }
```

Bloque de código A.4: Los arreglos son guardados de forma directa

Volver a descripción de la regla.

```
1 import java.security.MessageDigest;
2 public class AvoidMessageDigestFieldExample {
3     private final MessageDigest sharedMd;
4     public AvoidMessageDigestFieldExample() throws Exception {
5         sharedMd = MessageDigest.getInstance("SHA-256");
6     }
7     public byte[] calculateHashShared(byte[] data) {
8         // Compartir el campo MessageDigest de esta forma sin sincronización.
9         // podría generar resultados erróneos.
10        sharedMd.reset();
11        sharedMd.update(data);
12        return sharedMd.digest();
13    }
14
15    // Mejor seguir el patrón singleton:
16    public byte[] calculateHash(byte[] data) throws Exception {
```

```
17         MessageDigest md = MessageDigest.getInstance("SHA-256");
18         md.update(data);
19         return md.digest();
20     }
21 }
```

Bloque de código A.5: Regla Java: Evite el uso de la clase "*MessageDigest*" como un campo de clase

Volver a descripción de la regla.

```
1 class Foo {
2     void bar() {
3         try {
4             // haga algo
5         } catch (Exception e) {
6             e.printStackTrace(); //Incorrecto. use un Logger en su lugar.
7         }
8     }
9 }
```

Bloque de código A.6: Regla Java: Evite el uso de "*printStackTrace*"

Volver a descripción de la regla.

```
1 public class Foo {
2     public void foo() {
3         try {
4             // haga algo
5         } catch (Exception e) {
6             e = new NullPointerException(); // no es recomendado
7         }
8
9         try {
10            // haga algo
11        } catch (MyException | ServerException e) {
12            e = new RuntimeException(); // incompatibilidad de tipos lo cual
                 no permitirá que el código compile.
13        }
14    }
```

```
15 }
```

Bloque de código A.7: Regla Java: Evite reasignar variables en la sentencia de “*catch*”

Volver a descripción de la regla.

```
1  public class Foo {
2    private void foo() {
3      for (String s : listOfStrings()) {
4        s = s.trim(); // Se recomienda ser cuidadoso cuando se asigna la
            variable al inicio del ciclo.
5        doSomethingWith(s);
6
7        s = s.toUpper(); // Evitar asignaciones que pueden generar resultados
            inesperados
8        doSomethingElseWith(s);
9      }
10     for (int i=0; i < 10; i++) {
11       if (check(i)) {
12         i++; // Podría saltar iteraciones esperadas por la firma del ciclo.
13       }
14       i = 5;  // Podría generar un bucle infinito.
15       doSomethingWith(i);
16     }
17   }
18 }
```

Bloque de código A.8: Regla Java: Evite reasignar variables en ciclos

Volver a descripción de la regla.

```
1  public class Hello {
2    private void greet(String name) {
3      name = name.trim();
4      System.out.println("Hello " + name);
5
6      // preferido
7      String trimmedName = name.trim();
8      System.out.println("Hello " + trimmedName);
```

```
9     }
10  }
```

Bloque de código A.9: Regla Java: Evite la re-asignación de parámetros.

Volver a descripción de la regla.

```
1  public class Foo {
2      private StringBuffer buffer;    // Incorrecto. Potencial fuga de memoria
          al ser una variable de instancia.
3  }
```

Bloque de código A.10: Regla Java: Evite el uso de "*StringBuffer*"

Volver a descripción de la regla.

```
1  public class Foo {
2      private String ip = "127.0.0.1"; // Incorrecto. Lea esta propiedad desde
          un archivo de propiedades.
3  }
```

Bloque de código A.11: Regla Java: Evite la codificación fija de IPs

Volver a descripción de la regla.

```
1  Statement stat = conn.createStatement();
2  ResultSet rst = stat.executeQuery("SELECT name FROM person");
3  rst.next();     // qué pasaría si el siguiente resultado es vacío?
4  String firstName = rst.getString(1);
5  Statement stat = conn.createStatement();
6  ResultSet rst = stat.executeQuery("SELECT name FROM person");
7  if (rst.next()) {    // el resultado está siendo validado y usado
8      String firstName = rst.getString(1);
9  } else  {
10     // Agregué lógica para manejar resultados vacíos.
11 }
```

Bloque de código A.12: Regla Java: Valide los resultados obtenidos por un "*ResultSet*".

Volver a descripción de la regla.

```
1  public interface ConstantInterface {
```

```
2    public static final int CONST1 = 1; // Incorrecto. Los campos no son
        permitidos en interfaces.
3    static final int CONST2 = 1;        // Incorrecto. Los campos no son
        permitidos en interfaces.
4    final int CONST3 = 1;               // Incorrecto. Los campos no son
        permitidos en interfaces.
5    int CONST4 = 1;                     // Incorrecto. Los campos no son
        permitidos en interfaces.
6 }
```

Bloque de código A.13: Regla Java: Evite el uso de constantes en interfaces

Volver a descripción de la regla.

```
1  public class Foo {
2    void bar(int a) {
3     switch (a) {
4      case 1:
5         break;
6      default:  // El caso default debe estar al final por convención.
7         break;
8      case 2:
9         break;
10    }
11   }
```

Bloque de código A.14: Regla Java: El caso “*default*” debe estar de últimas en la sentencia de “*switch*”

Volver a descripción de la regla.

```
1 // Ejemplo de inicialización de doble llave.
2 return new ArrayList<String>(){{
3     add("a");
4     add("b");
5     add("c");
6 }};
7 // Es mejor seguir la siguiente convención para evitar la generación de clases
      anónimas.
8 List<String> a = new ArrayList<>();
```

```
9  a.add("a");
10 a.add("b");
11 a.add("c");
12 return a;
```

Bloque de código A.15: Regla Java: Evite la inicialización con doble llaves

Volver a descripción de la regla.

```
1  public class MyClass {
2    void loop(List<String> l) {
3      for (int i = 0; i < l.size(); i++) { // Incorrecto. Forma pre Java 1.5
4        System.out.println(l.get(i));
5      }
6
7      for (String s : l) { // Correcto. Forma post Java 1.5
8        System.out.println(s);
9      }
10   }
11 }
```

Bloque de código A.16: Regla Java: Ciclo "*For*" podría ser reemplazado por "*Foreach*"

Volver a descripción de la regla.

```
1 // Ejemplo de varias variables de control para un ciclo for.
2 for (int i = 0, j = 0; i < 10; i++, j += 2) {
3    foo();
```

Bloque de código A.17: Regla Java: Evite el uso de más de una variable de control para ciclos "*For*"

Volver a descripción de la regla.

```
1 // Agregué chequeo de nivel para incrementar el rendimiento.
2 if (log.isDebugEnabled()) {
3     log.debug("log something" + param1 + " and " + param2 + "concat strings");
4 }
5 // Ejemplo de chequeo mediante el uso de parámetros.
6 log.debug("log something {} and {}", param1, param2);
7 // Ejemplo de chequeo mediante el uso de formateadores
8 log.debug("log something %s and %s", param1, param2);
```

```
9  // Ejemplo de chequeo mediante el funciones lambda de registro perezoso.
10 log.debug("log something expensive: {}", () -> calculateExpensiveLoggingText()
   );
```

Bloque de código A.18: Regla Java: Valide el uso de un nivel de "*Log*"

Volver a descripción de la regla.

```
1 public class BadExample extends TestCase{
2     public static Test suite(){ // Incorrecto. prefiera la anotación @RunWith.
3         return new Suite();
4     }
5 }
6 @RunWith(Suite.class) // Correcto.
7 @SuiteClasses( { TestOne.class, TestTwo.class })
8 public class GoodTest {
9 }
```

Bloque de código A.19: Regla Java: Test de "*Junit 4*" deben usar la anotación "*RunWith*"

Volver a descripción de la regla.

```
1  public class MyTest {
2      public void tearDown() { //Incorrecto. Prefiera el uso de la anotación
           @After
3          bad();
4      }
5  }
6  public class MyTest2 {
7      @After public void tearDown() { // Correcto.
8          good();
9      }
10 }
```

Bloque de código A.20: Regla Java: Test de "*Junit 4*" deben usar la anotación "*After*"

Volver a descripción de la regla.

```
1 public class MyTest {
2     public void setUp() { // Incorrecto. Prefiera el uso de la anotación
          @Before
```

```
3           bad();
4       }
5   }
6   public class MyTest2 {
7       @Before public void setUp() { // Correcto.
8           good();
9       }
10  }
```

Bloque de código A.21: Regla Java: Test de “*Junit 4*” deben usar la anotación “*Before*”

Volver a descripción de la regla.

```
1   public class MyTest {
2       public void testBad() { // Incorrecto. Prefiera el uso de la anotación
            @Test.
3           doSomething();
4       }
5       @Test
6       public void testGood() {
7           doSomething();
8       }
9   }
```

Bloque de código A.22: Regla Java: Test de “*Junit 4*” deben usar la anotación “*Test*”.

Volver a descripción de la regla.

```
1   class MyTest {
2       @Test
3       public void testBad() { } // Incorrecto. No debe tener el modificador
            public
4       @Test
5       protected void testAlsoBad() { } // Incorrecto. No debe tener el
            modificador protected
6       @Test
7       private void testNoRun() { } // Incorrecto. No debe tener el modificador
            private
8       @Test
```

```
9       void testGood() { } // Correcto. encapsulamiento package como es esperado
10  }
```

Bloque de código A.23: Regla Java: Test de "*Junit 5*" debe usar encapsulamiento de tipo "*package*"

Volver a descripción de la regla.

```
1   public class Foo extends TestCase {
2       public void testSomething() {
3           assertEquals("foo", "bar"); // Incorrecto.
4           assertEquals("Foo does not equals bar", "foo", "bar"); // Correcto.
5       }
6   }
```

Bloque de código A.24: Regla Java: Aserciones de "*JUnit*" deben tener un mensaje

Volver a descripción de la regla.

```
1   public class MyTestCase extends TestCase {
2       // Correcto
3       public void testMyCaseWithOneAssert() {
4           boolean myVar = false;
5           assertFalse("should be false", myVar);
6           assertTrue("should be true", myVar);
7       }
8       // Incorrecto (asumiendo que el máximo de aserciones sea 2)
9       public void testMyCaseWithMoreAsserts() {
10          boolean myVar = false;
11          assertFalse("myVar should be false", myVar);
12          assertEquals("should equals false", false, myVar);
13          assertEquals("Foo does not equals bar", "foo", "bar");
14      }
15  }
```

Bloque de código A.25: Regla Java: test de JUnit contiene muchas aserciones

Volver a descripción de la regla.

```
1   public class Foo extends TestCase {
2       public void testSomething() {
3           Bar b = findBar();
```

```
4         // Es mejor tener una aserción que valide la nulabilidad
5         // de una variable en vez de tener un NullPointerExcepción
6         // assertNotNull("Variable bar no puede ser nulo.", b);
7         b.work();
8     }
9 }
```

Bloque de código A.26: Regla Java: Test de JUnit contener una aserción

Volver a descripción de la regla.

```
1  public class MyTest {
2      @Test
3      public void testBad() {
4          try {
5              doSomething();
6              fail("should have thrown an exception"); // Incorrecto. el método
                   doSomething lanza una excepción esta línea de código no se
                   puede alcanzar..
7          } catch (Exception e) {
8          }
9      }
10     @Test(expected=Exception.class) // Correcto. uso de expected cuando se
           espera una excepción.
11     public void testGood() {
12         doSomething(); // Método que lanza la excepción.
13     }
14 }
```

Bloque de código A.27: Regla Java: Test de JUnit debe contener la propiedad "*expected*" cuando se espera una excepción

Volver a descripción de la regla.

```
1  class Foo {
2      boolean bar(String x) {
3          return x.equals("2"); // Incorrecto. debería reemplazarse por "2".
               equals(x)
4      }
```

```
5      boolean bar(String x) {
6          return x.equalsIgnoreCase("2"); // Incorrecto. debería reemplazarse
               por "2".equalsIgnoreCase(x)
7      }
8      boolean bar(String x) {
9          return (x.compareTo("bar") > 0); // Incorrecto. debería reemplazarse
               por "bar".compareTo(x) < 0
10     }
11     boolean bar(String x) {
12         return (x.compareToIgnoreCase("bar") > 0); // Incorrecto. debería
               reemplazarse por "bar".compareToIgnoreCase(x) < 0
13     }
14     boolean bar(String x) {
15         return x.contentEquals("bar"); // Incorrecto. debería reemplazarse por
                "bar".contentEquals(x)
16     }
17 }
```

Bloque de código A.28: Regla Java: Los literales de "*String*" deben ir primero en las comparaciones

Volver a descripción de la regla.

```
1  import java.util.ArrayList;
2  import java.util.HashSet;
3  public class Bar {
4      // Incorrecto.
5      private ArrayList<SomeType> list = new ArrayList<>();
6      public HashSet<SomeType> getFoo() {
7          return new HashSet<SomeType>();
8      }
9      // Enfoque preferido.
10     private List<SomeType> list = new ArrayList<>();
11     public Set<SomeType> getFoo() {
12         return new HashSet<SomeType>();
13     }
14 }
```

Bloque de código A.29: Regla Java: Bajo acoplamiento

Volver a descripción de la regla.

```
1 public class SecureSystem {
2     UserData [] ud;
3     public UserData [] getUserData() {
4         // No retorne el arreglo interno. Retorne una copia del mismo.
5         return ud;
6     }
7 }
```

Bloque de código A.30: Regla Java: Evite el retorno de los arreglos internos

Volver a descripción de la regla.

```
1 public class Foo implements Runnable {
2     // Este método está siendo sobre-escrito y debería tener la anotación
         @Override.
3     public void run() {
4     }
5 }
```

Bloque de código A.31: Regla Java: Ausencia de la anotación "*Override*"

Volver a descripción de la regla.

```
1 String name;        // Correcto. Declaraciones separadas
2 String lastname;
3 String name,
4        lastname;    // Incorrecto. Declaraciones combinadas
```

Bloque de código A.32: Regla Java: Una declaración por línea

Volver a descripción de la regla.

```
1 public class Foo {
2     void good() {
3         try{
4             Integer.parseInt("a");
5         } catch (Exception e) {
6             throw new Exception(e); // Correcto. evita perder la excepción
                 original
```

```
7          }
8          try {
9              Integer.parseInt("a");
10         } catch (Exception e) {
11             throw (IllegalStateException)new IllegalStateException().initCause
                   (e); // Correcto. evita perder la exepción original.
12         }
13     }
14     void bad() {
15         try{
16             Integer.parseInt("a");
17         } catch (Exception e) {
18             throw new Exception(e.getMessage()); // Incorrecto. Se pierde la
                   cadena de excepciones.
19         }
20     }
21 }
```

Bloque de código A.33: Regla Java: Preservar "*Stack Trace*"

Volver a descripción de la regla.

```
1  public class Foo {
2      private Integer ZERO = new Integer(0);      // Incorrecto. Declarado
           obsoleto desde Java 9.
3      private Integer ZERO1 = Integer.valueOf(0); // Mejor.
4      private Integer ZERO1 = 0;                  // Opción preferida.
5  }
```

Bloque de código A.34: Regla Java: Uso de constructor para tipos primitivos

Volver a descripción de la regla.

```
1  public class Foo implements Enumeration { // Evite el uso de enumeration en la
       implementación de interfaces.
2      private int x = 42;
3      public boolean hasMoreElements() {
4          return true;
5      }
```

```
6      public Object nextElement() {
7          return String.valueOf(i++);
8      }
9  }
```

Bloque de código A.35: Regla Java: Reemplace el uso de “*Enumeration*” con “*Iterador*”

Volver a descripción de la regla.

```
1  public class Foo {
2      void bar() {
3          Hashtable h = new Hashtable(); // Incorrecto. Use Map m = new HashMap
               ();
4      }
5  }
```

Bloque de código A.36: Regla Java: Reemplace “*Hashtable*” con “*Map*”

Volver a descripción de la regla.

```
1  public class Foo {
2      void bar() {
3          Vector v = new Vector(); // Incorrecto. Use List l = new ArrayList();
4      }
5  }
```

Bloque de código A.37: Regla Java: Reemplace “*Vector*” con “*List*”

Volver a descripción de la regla.

```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3  class SomeTestClass {
4      Object a,b;
5      @Test
6      void testMethod() {
7          assertTrue(a.equals(b)); // Puede ser reemplazado por assertEquals(a,
               b);
8          assertTrue(!a.equals(b)); // Puede ser reemplazado por assertNotEquals
               (a, b);
```

```
9          assertTrue(!something); // Puede ser reemplazado por  assertFalse(
               something);
10         assertFalse(!something); // Puede ser reemplazado por  assertTrue(
               something);
11         assertTrue(a == b); // Puede ser reemplazado por assertSame(a, b);
12         assertTrue(a != b); // Puede ser reemplazado por  assertNotSame(a, b);
13         assertTrue(a == null); // Puede ser reemplazado por assertNull(a);
14         assertTrue(a != null); // Puede ser reemplazado por assertNotNull(a);
15     }
16 }
```

Bloque de código A.38: Regla Java: Simplifique el uso de aserciones

Volver a descripción de la regla.

```
1 class Foo {
2     int x = 2;
3     switch (x) {
4       case 1: int j = 6;
5       case 2: int j = 8;
6       // hace falta el caso default.
7     }
8 }
```

Bloque de código A.39: Regla Java: Sentencias "*switch*" deben tener un caso de "*default*"

Volver a descripción de la regla.

```
1 class Foo{
2     Logger log = Logger.getLogger(Foo.class.getName());
3     public void testA () {
4         System.out.println("Entrando al test"); // Incorrecto.
5         log.fine("Entrando al test"); // Correcto. Mejor use librerías de
              logging.
6     }
7 }
```

Bloque de código A.40: Regla Java: Evite el uso de "*System.(out|err).println*"

Volver a descripción de la regla.

```
// Ejemplo 1.
class A {
    // La inicialización es redundante ya que es sobre-escrita por el
        constructor.
    int f = 1;

    A(int f) {
        this.f = f;
    }
}
// Ejemplo 2
class B {
    int method(int i, int j) {
        // La inicialización es redundante, se sobre-escribe por ambas ramas
            del
        // condicional a continuación.
        int k = 0;
        if (i < j)
            k = i;
        else
            k = j;
        return j;
    }
}
// Ejemplo 3
class C {
    int method() {
        int i = 0;
        checkSomething(++i);
        checkSomething(++i);
        checkSomething(++i);
        checkSomething(++i);
        // La variable se incrementa antes de ser pasada como parámetro.
        // por lo tanto el valor previo nunca se lee.
    }
```

```
34 }
35 // Ejemplo 4
36 class C {
37     void method(int param) { } // el parámetro de este método nunca se lee.
38 }
```

Bloque de código A.41: Regla Java: Asignación no leída

Volver a descripción de la regla.

```
1 public class Foo {
2     private void bar(String howdy) {
3         // la variable howdy no es usada.
4     }
5 }
```

Bloque de código A.42: Regla Java: Parámetro no leído.

Volver a descripción de la regla.

```
1 public class Foo {
2     public void doSomething() {
3         int i = 5; // variable no usada.
4     }
5 }
```

Bloque de código A.43: Regla Java: Variable local no usada

Volver a descripción de la regla.

```
1 public class Something {
2     private static int FOO = 2; // No usado.
3     private int i = 5; // No usado.
4     private int j = 6;
5     public int addOne() {
6         return j++;
7     }
8 }
```

Bloque de código A.44: Atributo privado no usado

Volver a descripción de la regla.

```
public class Something {
    private void foo() {} // declarado pero no llamado dentro de la clase
        Something.
}
```

Bloque de código A.45: Regla Java: Método privado no usado

Volver a descripción de la regla.

```
public class Foo {
    void good() {
        List foo = getList();
        if (foo.isEmpty()) { // Correcto.
        }
    }
    void bad() {
        List foo = getList();
        if (foo.size() == 0) { // Incorrecto. No es necesario invocar el
            tamaño para comprobar si la colección no tiene elementos.
        }
    }
}
```

Bloque de código A.46: Regla Java: Uso de “*Collections.isEmpty()*”

Volver a descripción de la regla.

```
public class UseStandardCharsets {
    public void run() {
        // Incorrecto. Evite el uso de Charset.forName para conjuntos de
            caracteres comúnes.
        try (OutputStreamWriter osw = new OutputStreamWriter(out, Charset.
            forName("UTF-8"))) {
             osw.write("test");
        }
        // Correcto.
        try (OutputStreamWriter osw = new OutputStreamWriter(out,
            StandardCharsets.UTF_8)) {
             osw.write("test");
```

```
10          }
11      }
12  }
```

Bloque de código A.47: Regla Java: Use "*StandardCharsets*"

Volver a descripción de la regla.

```
1   public class TryWithResources {
2       public void run() {
3           InputStream in = null;
4           try {
5               in = openInputStream();
6               int i = in.read();
7           } catch (IOException e) {
8               e.printStackTrace();
9           } finally {
10              try {
11                  if (in != null) in.close();
12              } catch (IOException ignored) {
13                  // Incorrecto. excepción ignorada
14              }
15          }
16          // Esta notación se encarga del correcto manejo y cierre del recurso
                InputStream.
17          try (InputStream in2 = openInputStream()) {
18              int i = in2.read();
19          }
20      }
21  }
```

Bloque de código A.48: Regla Java: Use "*StandardCharsets*"

Volver a descripción de la regla.

```
1   public class Foo {
2       public void foo(String s, Object[] args) { // Incorrecto. No es necesaria
                la creación de un arreglo.
3       }
```

```
4     public void bar(String s, Object... args) { // Correcto. Esta sintaxis es
         más legible y denota el uso de argumentos variables.
5     }
6 }
```

Bloque de código A.49: Regla Java: Use argumentos variables “*varargs*”

Volver a descripción de la regla.

```
1  public class Example {
2    {
3      while (true) { } // Correcto.
4      while (false) { } // Incorrecto. El código dentro del alcance del while
          jamás será ejecutado.
5      do { } while (true); // Incorrecto. Obliga a leer toda la implementación
          del do para una condición siempre verdadera.
6      do { } while (false); // Incorrecto. No es necesario un bloque do|while
          para un ciclo que ejecutará una única vez.
7      do { } while (false | false); // Incorrecto. La condición siempre evalua
          falso.
8      do { } while (false || false); // Incorrecto. La condición siempre evalua
          falso.
9    }
10 }
```

Bloque de código A.50: Ciclo `while` con literal booleano

A.1.2. Ejemplos de estilo de código Java.

Volver a descripción de la regla.

```
1 public class Foo {
2    // no hay ningún constructor.
3   public void doSomething() { ... }
4   public void doOtherThing { ... }
5 }
```

Bloque de código A.51: Clase debe tener al menos un constructor.(PMD, 2022).

Volver a descripción de la regla.

```
1 public class Fo$o {  // Evite el símbolo de dinero
2 }
```

Bloque de código A.52: Evite el símbolo de dinero en el nombrado de variables o clases(PMD, 2022).

Volver a descripción de la regla.

```
1 public final class Bar {
2   private int x;
3   protected int y;  // Bar no se puede extender, por lo tanto no tiene sentido
        declarar atributos protegios o de tipo package.
4   Bar() {}
5 }
```

Bloque de código A.53: Evite campos protegidos o de tipo package en clases finales.

Volver a descripción de la regla.

```
1 public final class Foo {
2   private int bar() {}
3   protected int baz() {} // no se puede extender, por lo tanto no tiene
        sentido declarar métodos protegios o de tipo package.
4 }
```

Bloque de código A.54: Evite métodos protegidos o de tipo package en clases finales.

Volver a descripción de la regla.

```
1 public class SomeJNIClass {
2     public SomeJNIClass() {
3         System.loadLibrary("nativelib"); // Evite el uso de interfaces
              nativas JNI
4     }
5     static {
6         System.loadLibrary("nativelib"); // Evite el uso de interfaces
              nativas JNI
7     }
8     public void invalidCallsInMethod() throws SecurityException,
          NoSuchMethodException {
```

```
9           System.loadLibrary("nativelib"); // Evite el uso de interfaces
                nativas JNI
10      }
11  }
```

Bloque de código A.55: Evite el uso de código nativo.

Volver a descripción de la regla.

```
1  public boolean getFoo();             // Incorrecto
2  public boolean isFoo();              // Correcto
3  public boolean getFoo(boolean bar); // Correcto si y sólo si el método recibe
       un parámetro.
```

Bloque de código A.56: Evite el uso de get en métodos que retornan atributos de tipo `boolean`.

Volver a descripción de la regla.

```
1  public class Foo extends Bar{
2    public Foo() {
3     // Correcto. llame el constructor de Bar.
4     super();
5    }
6   public Foo(int code) {
7    // Correcto. llama el constructor por defecto que llama a super();
8     this();
9    }
10 }
```

Bloque de código A.57: Llame el constructor padre mediante el uso de `super`.

Volver a descripción de la regla.

```
1  // Correcto.
2  public class FooBar {}
3
4  // Incorrecto. Clases abstractas deben iniciar con el prefijo Abstract.
5  public abstract class Thing {}
6
7  // Incorrecto. Caracteres especiales no son permitidos en el nombrado de
       clases
```

```
8 public class Éléphant {}
```

Bloque de código A.58: Convenciones de nombrado de clase..

Volver a descripción de la regla.

```
1  public class Foo {
2      // Incorrecto. No se comenta el modificador de tipo package
3      final String stringValue = "some string";
4      // Incorrecto. No se comenta el modificador de tipo package
5      String getString() {
6          return stringValue;
7      }
8      // Incorrecto. No se comenta el modificador de tipo package
9      class NestedFoo {
10     }
11 }
12 // Correcto.
13 public class Foo {
14     /* default */ final String stringValue = "some string";
15     /* default */ String getString() {
16         return stringValue;
17     }
18
19     /* default */ class NestedFoo {
20     }
21 }
```

Bloque de código A.59: Comente el modificador por defecto de encapsulamiento.

Volver a descripción de la regla.

```
1 boolean bar(int x, int y) {
2     // Incorrecto. es confuso comparar el caso negativo primero.
3     return (x != y) ? diff : same;
4     // Correcto.
5     return (x == y) ? same : diff;
6 }
```

Bloque de código A.60: Evite el uso de operadores ternarios confusos.

Volver a descripción de la regla.

```
1 while (true)    // Incorrecto.
2   x++;
3 while (true) {  // Correcto.
4   x++;
5 }
```

Bloque de código A.61: Use paréntesis en sentencias de control .

Volver a descripción de la regla.

```
1 class Foo {
2     {
3         if (true); // Incorrecto. No posee ninguna sentencia adicional.
4         if (true) { // Incorrecto. No posee ninguna sentencia adicional.
5         }
6     }
7     {} // Inicializador vacío.
8 }
```

Bloque de código A.62: Sentencia de control vacía.

Volver a descripción de la regla.

```
1 public abstract class ShouldBeAbstract {
2     public Object couldBeAbstract() {
3         // ¿ Método debería ser abstracto?
4         return null;
5     }
6     public void couldBeAbstract() { // Incorrecto. los métodos vacíos deben
        ser abstractos en clases asbtractas.
7     }
8 }
```

Bloque de código A.63: Métodos vacíos de clases abstractas deben ser declarados como abstractos.

Volver a descripción de la regla.

```
1 public class Foo extends Object {     // No es necesario extender de Object.
2 }
```

Bloque de código A.64: No es necesario extender `Object`.

Volver a descripción de la regla.

```
1 public class HelloWorldBean {
2   // Correcto. Atributo declarado antes de los métodos.
3   private String _thing;
4   public String getMessage() {
5     return "Hello World!";
6   }
7   // Incorrecto. atributo declarado después de los métodos.
8   private String _fieldInWrongLocation;
9 }
```

Bloque de código A.65: Declare los atributos antes de los métodos.

Volver a descripción de la regla.

```
1 class Foo {
2     int myField = 1; // Correcto. respeta la convención 'camelCase'
3     int my_Field = 1; // Incorrecto. contiene guión bajo lo cual no es común
        para Java.
4     final int FinalField = 1; // Incorecto. no respeta la convención '
        camelCase'
5     interface Interface {
6         double PI = 3.14; // Incorrecto. las constantes deben ser declaradas
            con mayúsculas, separadas por guiones bajos.
7     }
8     enum AnEnum {
9         ORG, NET, COM; //Correcto.
10    }
11 }
```

Bloque de código A.66: Convenciones de nombrado de atributos.

Volver a descripción de la regla.

```
1 public interface MyInterface {
2   void process(final Object arg); // Evite el uso de final.
```

```
3 }
```

Bloque de código A.67: Evite el uso de constantes mediante el modificador `final` como parámetros en métodos de clases abstractas.

Volver a descripción de la regla.

```
1 public class Foo {
2     void bar() {
3         for (;true;) true; // No es necesario usar esta forma cuando puede ser
              reemplazado por while(true).
4     }
5 }
```

Bloque de código A.68: Ciclo de tipo `for` debería ser `while`.

Volver a descripción de la regla.

```
1 class Foo {
2     abstract void bar(int myInt); // Correcto. no respeta la convención de '
         camelCase'
3     void bar(int my_i) { // Incorrecto. no respeta la convención de 'camelCase
         '
4     }
5     void lambdas() {
6         // Incorrecto. no respeta la convención de 'camelCase'
7         Consumer<String> lambda1 = s_str -> { };
8         // Correcto. no respeta la convención de 'camelCase'
9         Consumer<String> lambda1 = (String str) -> { };
10    }
11 }
```

Bloque de código A.69: Convención de nombrado de parámetros.

Volver a descripción de la regla.

```
1 public interface GenericDao<E extends BaseModel, K extends Serializable>
     extends BaseDao {
2     // Correcto. Letra mayúscula única.
3 }
```

```
4  public interface GenericDao<E extends BaseModel, K extends Serializable> {
5      // Correcto. Letra mayúscula única.
6  }
7  public interface GenericDao<e extends BaseModel, K extends Serializable> {
8      // Incorrecto. 'e' debería ser 'E'
9  }
10 public interface GenericDao<EF extends BaseModel, K extends Serializable> {
11    // Incorrecto. 'EF' contiene dos letras mayúsculas.
12 }
```

Bloque de código A.70: Convención de nombrado de Genéricos.

Volver a descripción de la regla.

```
1  try {
2  } catch (IllegalArgumentException e) {
3      throw e;
4  } catch (IllegalStateException e) { // Incorrecto. Esta rama hace lo mismo que
       la anterior podrían ser unidas.
5      throw e;
6  }
7  try {
8  } catch (IllegalArgumentException | IllegalStateException e) { // Correcto.
     las ramas pueden ser unidas en una con el operador or '|'
9      throw e;
10 }
```

Bloque de código A.71: Ramas de `catch` idénticas.

Volver a descripción de la regla.

```
1  public class LinguisticNaming {
2      int isValid;    // Incorrecto. el nombre de la variable indica que es un
          booleano, pero la variable es de tipo entero.
3      boolean isTrue; // Correcto. Nombre y tipo son coherentes.
4      void myMethod() {
5          int hasMoneyLocal;      // Incorrecto. el nombre de la variable indica
               que es un booleano, pero la variable es de tipo entero.
6          boolean hasSalaryLocal; // Correcto. Nombre y tipo son coherentes.
```

```
7       }
8       // Incorrecto. el nombre del método indica el retorno de un booleano, pero
            retorna un entero.
9       int isValid() {
10          return 1;
11      }
12      // Correcto. Nombre y tipo son coherentes.
13      boolean isSmall() {
14          return true;
15      }
16      // Incorrecto. el nombre del método indica un procedimiento de asignación,
            no obstante retorna un entero.
17      int setName() {
18          return 1;
19      }
20      // Incorrecto. El nombre del método indica que se obtiene un valor, no
           obstante es un método de retorno vacío
21      void getName() {
22          // No se retorna nada.
23      }
24      // Incorrecto. El nombre indica una transformación, no obstante no se
           retorna nada.
25      void toDataType() {
26          // No se retorna nada.
27      }
28  }
```

Bloque de código A.72: Nombre semántico coherente.

Volver a descripción de la regla.

```
1 public interface MyBeautifulLocalHome extends javax.ejb.EJBLocalHome {} //
     Correcto.
2 public interface MissingProperSuffix extends javax.ejb.EJBLocalHome {}  //
     Incorrecto.
```

Bloque de código A.73: Nombre coherente para “*Beans*” de tipo sesión `LocalHome`.

Volver a descripción de la regla.

```
1 public interface MyLocal extends javax.ejb.EJBLocalObject {}           //
      Correcto. Posee el literal 'Local'
2 public interface MissingProperSuffix extends javax.ejb.EJBLocalObject {}    //
      Incorrecto. no posee el literal 'Local'
```

Bloque de código A.74: Nombre coherente para "*Interfaces*" de sesión de tipo `Local`.

Volver a descripción de la regla.

```
1 public class Bar {
2     public void foo () {
3     String txtA = "a";           // Incorrecto. 'txtA' debe ser final toda vez
          su valor no cambia.
4     final String txtB = "b";     // Correcto.
5     }
6 }
```

Bloque de código A.75: Variables locales deben ser de tipo `final`.

Volver a descripción de la regla.

```
1 class Foo {
2     void bar() {
3         int localVariable = 1; // Correcto. respeta la convención 'camelCase'
4         int local_variable = 1; // Incorrecto.
5         final int i_var = 1; // Incorrecto.
6         try {
7             foo();
8         } catch (IllegalArgumentException e_illegal) { // Incorrecto.
9         }
10    }
11 }
```

Bloque de código A.76: Convención de nombrado de variables locales.

Volver a descripción de la regla.

```
1 public class Something {
2     int reallyLongIntName = -3;              // Incorrecto. Atributo.
```

```
3     public static void main( String argumentsList[] ) { // Incorrecto.
          parámetro de metodo.
4         int otherReallyLongName = -5;       // Incorrecto. variable local.
5         for (int interestingIntIndex = 0;   // Incorrecto. variable de ciclo
              for.
6              interestingIntIndex < 10;
7              interestingIntIndex ++ ) {
8     }
9 }
```

Bloque de código A.77: Evite el nombrado de variables de más de 17 caracteres.

Volver a descripción de la regla.

```
1 public class SomeBean implements SessionBean{}                    // Correcto.
      Posee el sufijo Bean
2 public class MissingTheProperSuffix implements SessionBean {}    // Incorrecto.
```

Bloque de código A.78: Convención para nombrado de *"Beans"* de tipo `MessageDrivenBean` o `SessionBean`.

Volver a descripción de la regla.

```
1 public void foo2 (final String param) { // Declare el parámetro como final si
      no se cambia su valor.
2
3 }
```

Bloque de código A.79: Parámetro formal puede ser `final`.

Volver a descripción de la regla.

```
1 public class Foo {
2     public void fooStuff() { // Correcto. Respeta la convención 'camelCase'.
3     }
4 }
```

Bloque de código A.80: Convención para el nombrado de métodos.

Volver a descripción de la regla.

```
1 // Incorrecto. No hay declaración de paquete.
```

```
2 public class ClassInDefaultPackage {
3 }
```

Bloque de código A.81: Clase no pertenece a ningún paquete.

Volver a descripción de la regla.

```
1 public class OneReturnOnly1 {
2   public String foo(int x) {
3     if (x > 0) {
4       return "hey"; // Incorrecto. si no existe otra sentencia de retorno.
5     }
6     return "hi"; // Correcto.
7   }
8 }
```

Bloque de código A.82: Método debe tener retorno al final del método.

Volver a descripción de la regla.

```
1 package com.MyCompany;  // sólo debe contener caracteres en minúsculas.
2 public class SomeClass {
3 }
```

Bloque de código A.83: Convención de nombrado de paquetes.

Volver a descripción de la regla.

```
1 public int getLength(String[] strings) {
2     int length = 0; // Variable declarada mucho antes de ser usada.
3     if (strings == null || strings.length == 0) return 0;
4     for (String str : strings) {
5         length += str.length();
6     }
7     return length;
8 }
```

Bloque de código A.84: Declaración prematura de variuables.

Volver a descripción de la regla.

```
1 /* Incorrecto. No posee el suifijo 'SessionBean' */
```

```
2 public interface BadSuffixSession extends javax.ejb.EJBObject {}
3 /* Incorrecto. No posee el suifijo 'SessionBean' */
4 public interface BadSuffixEJB extends javax.ejb.EJBObject {}
5 /* Incorrecto. No posee el suifijo 'SessionBean' */
6 public interface BadSuffixBean extends javax.ejb.EJBObject {}
```

Bloque de código A.85: Convención para el nombrado de "*Beans*" EJB.

Volver a descripción de la regla.

```
1 public interface MyBeautifulHome extends javax.ejb.EJBHome {}        //
    Correcto. Posee el sufijo Home
2 public interface MissingProperSuffix extends javax.ejb.EJBHome {}   //
    Incorrecto.
```

Bloque de código A.86: Convención para el nombrado de "*EJBs*" de sesión locales remotos.

Volver a descripción de la regla.

```
1 public class Foo { // Incorrecto. los nombres de las clases deben tener al
    menos cinco caracteres.
2 }
```

Bloque de código A.87: Nombre de clase muy corto.

Volver a descripción de la regla.

```
1 public class ShortMethod {
2     public void a( int i ) { // Incorrecto. Los nombres de los métodos deben
        tener al menos tres caracteres.
3     }
4 }
```

Bloque de código A.88: Nombre de método muy corto.

Volver a descripción de la regla.

```
1 public class Something {
2     private int q = 15;                          // Muy corto.
3     public static void main( String as[] ) {     // Muy corto.
4         int r = 20 + q;                          // Muy corto.
5         for (int i = 0; i < 10; i++) {           // No aplica para for.
```

```
            r += q;
        }
        for (Integer i : numbers) {              // No aplica para for.
            r += q;
        }
    }
}
```

Bloque de código A.89: Nombre de variable muy corto.

Volver a descripción de la regla.

```
import static Lennon;
import static Ringo;
import static George;
import static Paul;
import static Yoko; // Máximo 4 imports estáticos.
```

Bloque de código A.90: Demasiados `import` estáticos.

Volver a descripción de la regla.

```
// Incorrecto. no es necesario agregar el literal value en las anotaciones.
@TestClassAnnotation(value = "TEST")
public class Foo {
    @TestMemberAnnotation(value = "TEST")
    private String y;
    @TestMethodAnnotation(value = "TEST")
    public void bar() {
        int x = 42;
        return;
    }
}
// Correcto.
@TestClassAnnotation("TEST")
public class Foo {
    @TestMemberAnnotation("TEST")
    private String y;
    @TestMethodAnnotation("TEST")
```

```
18      public void bar() {
19          int x = 42;
20          return;
21      }
22  }
```

Bloque de código A.91: Literal `value` innecesario en anotaciones.

Volver a descripción de la regla.

```
1  public class UnnecessaryCastSample {
2      public void method() {
3          List<String> stringList = Arrays.asList("a", "b");
4          String element = (String) stringList.get(0); // Conversión innecesaria
               , la lista posee argumentos de tipo String.
5          String element2 = stringList.get(0);
6      }
7  }
```

Bloque de código A.92: “*Casting*”/Conversión innecesaria.

Volver a descripción de la regla.

```
1  public class Foo {
2    public Foo() {} // Incorrecto. No es necesario declarar un constructor vacío
       .
3  }
```

Bloque de código A.93: Constructor innecesario.

Volver a descripción de la regla.

```
1  import java.util.List;
2  public class Foo {
3      private java.util.List list1;   // Incorrecto. no es necesario usar el
          nombre completamente calificado, debido a que ya se está importando la
           lista.
4      private List list2;             // Correcto.
5  }
```

Bloque de código A.94: Innecesario uso de nombre completamente calificado.

Volver a descripción de la regla.

```
1  // Correcto.
2  import java.util.Collections;
3  // Incorrecto. No es necesario, debido a que la clase no es usada.
4  import java.io.File;
5  // Incorrecto. No se usa una clase diferente a Collections.
6  import java.util.*;
7  // Incorrecto. No es necesario importar clases del paquete java.lang
8  import java.lang.Object;
9  // Incorrecto. Importación duplicada.
10 import java.util.Collections;
11 public class Foo {
12     static Object emptyList() {
13         return Collections.emptyList();
14     }
15 }
```

Bloque de código A.95: Importación innecesaria.

Volver a descripción de la regla.

```
1 public class Foo {
2    public int foo() {
3      int x = doSomething(); // Incorrecto. Innecesario, simplemente retorne '
          return doSomething();'
4      return x;
5    }
6 }
```

Bloque de código A.96: Variable local innecesaria para retorno.

Volver a descripción de la regla.

```
1 public @interface Annotation {
2     public abstract void bar();     // Incorrecto. abstract y public son
         ignorados por el compilador
3     public static final int X = 0;  // Incorrecto. public, static, y final son
          ignorados por el compilador.
```

```
4       public static class Bar {}      // Incorrecto. public, static son
           ignorados por el compilador.
5       public static interface Baz {}  // Incorrecto. public, static son
           ignorados por el compilador.
6   }
7   public class Bar {
8       public static interface Baz {}  // Incorrecto. static es ignorado por el
           compilador.
9       public static enum FoorBar {    // Incorrecto. static es ignorado por el
           compilador.
10          FOO;
11      }
12  }
13  public class FooClass {
14      static record BarRecord() {}     // Incorrecto. static es ignorado por el
           compilador.
15  }
16  public interface FooInterface {
17      static record BarRecord() {}     // Incorrecto. static es ignorado por el
           compilador.
18  }
```

Bloque de código A.97: Modificador innecesario.

Volver a descripción de la regla.

```
1   public class Foo {
2       public void bar() {
3           int x = 42;
4           return; // Incorrecto. retorno innecesario.
5       }
6   }
```

Bloque de código A.98: Retorno innecesario.

Volver a descripción de la regla.

```
1   class Foo {
2       {
```

```
3         toString();; // Incorrecto. Uno de los dos punto y comas es
             innecesario.
4         if (true); // Incorrecto. El punto y coma no es innecesario, pero es
             mejor reemplazarlo por un bloque vacío
5     }
6 }; // Incorrecto. punto y coma innecesario.
```

Bloque de código A.99: Punto y coma innecesario.

Volver a descripción de la regla.

```
1 List<String> strings = new ArrayList<String>(); // Incorrecto. Innecesaria
     especificación de tipo en el ArrayList
2 List<String> stringsWithDiamond = new ArrayList<>(); //  Correcto. Uso del
     operador de diamante.
```

Bloque de código A.100: Use el operador diamante `'<>'`.

Volver a descripción de la regla.

```
1 public class Foo {
2     private int _bar1;
3     private Integer _bar2;
4     public void setBar(int n) {
5         _bar1 = Integer.valueOf((n)); // Incorrecto. paréntesis innecesario.
6         _bar2 = (n); // Incorrecto. paréntesis innecesario.
7     }
8 }
```

Bloque de código A.101: Paréntesis inutil.

Volver a descripción de la regla.

```
1 public class Foo {
2     final Foo otherFoo = Foo.this;  // Incorrecto. use "this" directamente.
3     public void doSomething() {
4          final Foo anotherFoo = Foo.this;  // Incorrecto. use "this"
              directamente.
5     }
6     private ActionListener returnListener() {
7         return new ActionListener() {
```

```
8           @Override
9           public void actionPerformed(ActionEvent e) {
10              doSomethingWithQualifiedThis(Foo.this);  // Correcto.
11          }
12      };
13  }
14  private class Foo3 {
15      final Foo myFoo = Foo.this;  // Correcto.
16  }
17  private class Foo2 {
18      final Foo2 myFoo2 = Foo2.this;  // Incorrecto. use "this" directamente
            .
19  }
20 }
```

Bloque de código A.102: Uso de calificador `this` innecesario.

Volver a descripción de la regla.

```
1 Foo[] x = new Foo[] { ... }; // Incorrecto. Verborrea innecesaria.
2 Foo[] x = { ... }; //Correcto. equivalente a la sentencia anterior.
```

Bloque de código A.103: Use inicializador de arreglos corto.

Volver a descripción de la regla.

```
1 public class Foo {
2     private int num = 1000000; // Use 1_000_000 para mejorar la legibilidad
          del número.
3 }
```

Bloque de código A.104: Use guión bajos en literales numéricos.

A.1.3. Ejemplos de reglas de diseño Java.

Volver a descripción de la regla.

```
1 public abstract class Example {
2     String field;
```

```
3     int otherField;
4     // No hay ningún método en la clase abstracta.
5 }
```

Bloque de código A.105: Clase abstracta sin ningún método.

Volver a descripción de la regla.

```
1  package com.igate.primitive;
2  public class PrimitiveType {
3      public void downCastPrimitiveType() {
4          try {
5              System.out.println(" i [" + i + "]");
6          } catch(Exception e) { // Incorrecto. el bloque try no lanza
               excepciones genéricas.
7              e.printStackTrace();
8          } catch(RuntimeException e) { // Incorrecto. el bloque try no lanza
               excepciones genéricas.
9              e.printStackTrace();
10         } catch(NullPointerException e) { // Incorrecto. el bloque try no
               lanza NullPointerException
11             e.printStackTrace();
12         }
13     }
14 }
```

Bloque de código A.106: Evite capturar excepciones genéricas.

Volver a descripción de la regla.

```
1 public class Foo {
2   public void bar(int x, int y, int z) {
3     if (x>y) {
4       if (y>z) {
5         if (z==x) {
6          // Incorrecto. Evite tener más de dos condicionales anidados.
7         }
8       }
9     }
```

```
10    }
11 }
```

Bloque de código A.107: Evite el exceso de condicionales anidados.

Volver a descripción de la regla.

```
1 public void bar() {
2     try {
3     }  catch (SomeException se) {
4        throw se; // Incorrecto. Evite el relanzamiento de la misma excepción,
            quizás debe reemplazarse por un throws en la firma del método.
5     }
6 }
```

Bloque de código A.108: Evite el relanzamiento de excepciones.

Volver a descripción de la regla.

```
1 public void bar() {
2     try {
3     } catch (SomeException se) {
4         // Incorrecto. evite el relanzamiento de la misma excepción.
5         throw new SomeException(se);
6     }
7 }
```

Bloque de código A.109: Evite el lanzamiento de una nueva instancia de la misma excepción.

Volver a descripción de la regla.

```
1 public class Foo {
2     void bar() {
3         throw new NullPointerException(); // Incorrecto. Evite el lanzamiento
             de NullPointerException
4     }
5 }
```

Bloque de código A.110: Evite el lanzamiento de la excepción de `NullPointerException`.

Volver a descripción de la regla.

```
1 public class Foo {
2     void bar() {
3         throw new Exception(); // Incorrecto. Use una excepción más específica
              .
4     }
5 }
```

Bloque de código A.111: Evite el lanzamiento de excepciones genéricas.

Volver a descripción de la regla.

```
1 public void foo() throws RuntimeException { // Incorrecto. Prefiera el uso de
      excepciones chequeadas.
2 }
```

Bloque de código A.112: Evite el lanzamiento de excepciones no chequeadas.

Volver a descripción de la regla.

```
1 public class Foo {  //Incorrecto. la clase debe ser final.
2     private Foo() { }
3 }
```

Bloque de código A.113: Clases con constructores privadas deben ser de tipo `final`.

Volver a descripción de la regla.

```
1  public class Foo {
2    // Posee una complejidad cognitiva de 0
3    public void createAccount() {
4      Account account = new Account("PMD");
5      // save account
6    }
7    // Posee una complejidad cognitiva de 1
8    public Boolean setPhoneNumberIfNotExisting(Account a, String phone) {
9      if (a.phone == null) {                          // +1
10       a.phone = phone;
11       return true;
12     }
13     return false;
14   }
```

```
15   // Posee una complejidad cognitiva de 4
16   public void updateContacts(List<Contact> contacts) {
17     List<Contact> contactsToUpdate = new ArrayList<Contact>();
18     for (Contact contact : contacts) {                          // +1
19       if (contact.department.equals("Finance")) {               // +2 (
         anidado = 1)
20         contact.title = "Finance Specialist";
21         contactsToUpdate.add(contact);
22       } else if (contact.department.equals("Sales")) {          // +1
23         contact.title = "Sales Specialist";
24         contactsToUpdate.add(contact);
25       }
26     }
27   }
28 }
```

Bloque de código A.114: Complejidad cognitiva.

Volver a descripción de la regla.

```
1  void bar() {
2      if (x) {            // Implementación original.
3          if (y) {
4          }
5      }
6  }
7  void bar() {
8      if (x && y) {        // Implementación optimizada.
9      }
10 }
```

Bloque de código A.115: Condicionales agrupables.

Volver a descripción de la regla.

```
1 import com.Blah;
2 import org.Bar;
3 import org.Bardo;
4 public class Foo {
```

```
5      private Blah var1;
6      private Bar var2;
7      // Importación de objetos únicos.
8      ObjectC doWork() {
9          Bardo var55;
10         ObjectA var44;
11         ObjectZ var93;
12         return something();
13     }
14 }
```

Bloque de código A.116: Acoplamiento entre objetos.

Volver a descripción de la regla.

```
1  class Foo {
2    void baseCyclo() {                // Ciclomática = 1
3      highCyclo();
4    }
5    void highCyclo() {                // Incorrecto. Ciclomática = 10 !
6      int x = 0, y = 2;
7      boolean a = false, b = true;
8      if (a && (y == 1 ? b : true)) { // +3
9        if (y == x) {                 // +1
10         while (true) {              // +1
11           if (x++ < 20) {           // +1
12             break;                  // +1
13           }
14         }
15       } else if (y == t && !d) {    // +2
16         x = a ? y : x;              // +1
17       } else {
18         x = 2;
19       }
20     }
21   }
```

```
22 }
```

Bloque de código A.117: Complejidad ciclomática.

Volver a descripción de la regla.

```
1  public class DataClass {
2    // Atributos públicos expuestos.
3    public String name = "";
4    public int bar = 0;
5    public int na = 0;
6    private int bee = 0;
7    // Atributos privados son expuestos mediante 'getters y setters'
8    public void setBee(int n) {
9      bee = n;
10   }
11 }
```

Bloque de código A.118: Clase de tipo `Data`

Volver a descripción de la regla.

```
1  public class Foo extends Error { } // Incorrecto. no exitenda Error
```

Bloque de código A.119: No extienda `java.lang.Error`

Volver a descripción de la regla.

```
1  public void bar() {
2      try {
3          try {
4          } catch (Exception e) {
5              throw new WrapperException(e); // Incorrecto. Se está lanzando la
                  excepción WrapperException para ir al bloque catch.
6          }
7      } catch (WrapperException e) {
8      }
9  }
```

Bloque de código A.120: No use las excepciones como flujo de control.

Volver a descripción de la regla.

```java
import blah.blah.Baz;
import blah.blah.Bif;
// 28 otras importaciones al mismo paquete blah.blah
public class Foo {
    public void doWork() {}
}
```

Bloque de código A.121: Importaciones excesivas.

Volver a descripción de la regla.

```java
public void addPerson(      // Incorrecto. muchos atributos pueden ser confundidos al momento del llamado.
    int birthYear, int birthMonth, int birthDate, int height, int weight, int ssn) {
    . . .
}
public void addPerson(      // Correcto. Mejor declare un objeto que encapsule la gran cantidad de atributos recibidos por el método.
    Date birthdate, BodyMeasurements measurements, int ssn) {
    . . .
}
```

Bloque de código A.122: Número de parámetros excesivo.

Volver a descripción de la regla.

```java
public class Foo {
    // Evite usar más de 45 elementos con visibilidad pública.
    public String value;
    public Bar something;
    public Variable var;
    // [... más atributos publicos ...]
    public void doWork() {}
    public void doMoreWork() {}
    public void doWorkAgain() {}
    // [... más métodos publicos....]
}
```

Bloque de código A.123: Número de elementos públicos excesivo.

Volver a descripción de la regla.

```
1 public class Foo {
2   public final int BAR = 42; // Incorrecto. atributo puede ser estático para
      ahorrar espacio.
3 }
```

Bloque de código A.124: Atributo `final` puede ser `static`.

Volver a descripción de la regla.

```
1 public class Foo {
2   private int x; // Incorrecto. atributo no cambia por lo tanto puede ser
      final.
3   public Foo() {
4       x = 7;
5   }
6   public void foo() {
7      int a = x + 2;
8   }
9 }
```

Bloque de código A.125: Atributo inmutable puede ser `final`.

Volver a descripción de la regla.

```
1 package org.example.beans;
2 public class MyBean {        // <-- Bean no implementa serializable = "
     implements Serializable"
3     private String label;    // <-- Setter faltante para el atributo "label"
4     public String getLabel() {
5         return label;
6     }
7 }
```

Bloque de código A.126: “*Bean*” de java inválido.

Volver a descripción de la regla.

```
1 public class Foo {
2     /**
```

```
3      * Este ejemplo muestra 2 violaciones de la ley de Demeter = 'Principio de
           menor conocimiento'
4      */
5     public void example(Bar b) {
6         // Correcto. b es un parámetro del método "example".
7         C c = b.getC();
8         // Incorrecto. Este llamado es una violación de la regla puesto que en
              vez de obtener C se pudo llamar desde b directamente "b.doItOnC()
             ;"
9         c.doIt();
10        // Incorrecto. Llamado en cadena también genera que esta clase conozca
              más de b de lo que se requiere.
11        b.getC().doIt();
12        D d = new D();
13        // Correcto. La llamada es correcta debido toda vez que se crea una
             instancia de D directamente.
14        d.doSomethingElse();
15    }
16 }
```

Bloque de código A.127: Ley de “*Demeter*”.

Volver a descripción de la regla.

```
1 public boolean bar(int a, int b) {
2     if (!(a == b)) { // Incorrecto. Complejidad innecesaria, reemplace la
         sentencia por 'a != b'
3          return false;
4      }
5     if (!(a < b)) { // Incorrecto. Complejidad innecesaria, reemplace por 'a
         >= b'
6          return false;
7     }
8     return true;
9 }
```

Bloque de código A.128: Inversión de lógica innecesaria.

Volver a descripción de la regla.

```
1 package some.package;
2 import some.other.package.subpackage.subsubpackage.DontUseThisClass;
3 public class Bar {
4     DontUseThisClass boo = new DontUseThisClass(); // Incorrecto. evite el
        alto acoplamiento entre paquetes de jerarquía común.
5 }
```

Bloque de código A.129: Evite el alto acoplamiento entre clases de la misma jerarquía de paquetes.

Volver a descripción de la regla.

```
1 public class Greeter { public static Foo foo = new Foo(); ... }       //
    Incorrecto. Evite esto.
2 public class Greeter { public static final Foo foo = new Foo(); ... } //
    Correcto. el estado del atributo foo es final.
```

Bloque de código A.130: Estado estático mutable.

Volver a descripción de la regla.

```
1  import java.util.Collections;       // +0
2  import java.io.IOException;         // +0
3  class Foo {                         // +1, total = 12
4
5    public void bigMethod()           // +1
6        throws IOException {
7      int x = 0, y = 2;               // +1
8      boolean a = false, b = true;    // +1
9
10     if (a || b) {                   // +1
11       try {                         // +1
12         do {                        // +1
13           x += 2;                   // +1
14         } while (x < 12);
15
16         System.exit(0);             // +1
17       } catch (IOException ioe) {   // +1
18         throw new PatheticFailException(ioe); // +1
```

```
        }
      } else {                        // +1
        assert false;                 // +1
      }
    }
}
```

Bloque de código A.131: Metrica de conteo de sentencias.

Volver a descripción de la regla.

```
public class Foo {
  public static void bar() { // 252 caminos reportados!
    boolean a, b = true;
    try { // 2 * 2 + 2 = 6
      if (true) { // 2
        List buz = new ArrayList();
      }
      for(int i = 0; i < 19; i++) { // * 2
        List buz = new ArrayList();
      }
    } catch(Exception e) {
      if (true) { // 2
        e.printStackTrace();
      }
    }
    while (j++ < 20) { //  * 2
      List buz = new ArrayList();
    }
    switch(j) { // * 7
      case 1:
      case 2: break;
      case 3: j = 5; break;
      case 4: if (b && a) { bar(); } break;
      default: break;
    }
    do { // * 3
```

```
27          List buz = new ArrayList();
28      } while (a && j++ < 30);
29    }
30  }
```

Bloque de código A.132: Metrica de conteo de distintos caminos.

Volver a descripción de la regla.

```
1  public void foo() throws Exception { // No es claro el tipo de error que
       Exception encapsula.
2  }
```

Bloque de código A.133: Firma de método lanza `Exception`.

Volver a descripción de la regla.

```
1  public class Foo {
2      public boolean test() {
3          return condition ? true : something(); // Incorrecto. no es necesario
               usar un operador ternario, puede ser transformado por "return
               condition || something();"
4      }
5      public void test2() {
6          final boolean value = condition ? false : something(); // Incorrecto.
               no es necesario usar un operador ternario, puede ser transformado
               por "value = !condition && something()";
7      }
8      public boolean test3() {
9          return condition ? something() : true; // Incorrecto. no es necesario
               usar un operador ternario, puede ser transformado por "return !
               condition || something();"
10     }
11     public void test4() {
12         final boolean otherValue = condition ? something() : false; ///
               Incorrecto. no es necesario usar un operador ternario, puede ser
               transformado por "condition && something();"
13     }
14     public boolean test5() {
```

```
15          return condition ? true : false; // // Incorrecto. no es necesario
                usar un operador ternario, puede ser transformado por "return
                condition;"
16      }
17  }
```

Bloque de código A.134: Operador ternario simplificado.

Volver a descripción de la regla.

```
1  public class Bar {
2    // puede ser reemplazado por "bar = isFoo()"
3    private boolean bar = (isFoo() == true);
4
5    public isFoo() { return false;}
6  }
```

Bloque de código A.135: Operadores Booleanos simplificados.

Volver a descripción de la regla.

```
1  public boolean isBarEqualTo(int x) {
2      if (bar == x) {      // Incorrecto. Verborrea innecesaria.
3          return true;
4      } else {
5          return false;
6      }
7  }
8  public boolean isBarEqualTo(int x) {
9      return bar == x;    // Correcto.
10 }
```

Bloque de código A.136: Retornos Booleanos simplificados.

Volver a descripción de la regla.

```
1  class Foo {
2    void bar(Object x) {
3      if (x != null && x instanceof Bar) { // la validación "x != null" es
           innecesaria.
4      }
```

```
5     }
6 }
```

Bloque de código A.137: Condicionales simplificados.

Volver a descripción de la regla.

```
1 public class Foo {
2     private int x;  // Incorrecto. no es necesario que la variable x sea un
         atributo de clase.
3     public void foo(int y) {
4      x = y + 5;
5      return x;
6     }
7 }
```

Bloque de código A.138: Atributo singular.

Volver a descripción de la regla.

```
1 public class Foo {
2   public void bar(int x) {
3     switch (x) {
4       case 1: {
5         // Más de 10 sentencias. Extraerlas a otro método.
6         break;
7       } case 2: {
8         // Más de 10 sentencias. Extraerlas a otro método.
9         break;
10      }
11    }
12  }
13 }
```

Bloque de código A.139: Densidad en bloque `switch`.

Volver a descripción de la regla.

```
1 public class Person {   // Más de 15 atributos son más complejos de mantener.
2    int birthYear;
3    int birthMonth;
```

```
4     int birthDate;
5     float height;
6     float weight;
7  }
8  public class Person {   // Es mejor extraer atributos comunes a otros objetos
9     Date birthDate;
10    BodyMeasurements measurements;
11 }
```

Bloque de código A.140: Demasiados atributos para una clase.

Volver a descripción de la regla.

```
1  public void foo(String bar) {
2      super.foo(bar);      // No es necesario sobreescribir el método.
3  }
4  public String foo() {
5      return super.foo();  // No es necesario sobreescribir el método.
6  }
```

Bloque de código A.141: Sobreescritura de métodos innecesaria.

Volver a descripción de la regla.

```
1  public class MyClass {
2      public void connect(String username,
3          String pssd,
4          String databaseName,
5          String databaseAdress) // Extraiga los atributos a una clase 'UserData
           ' para asegurar un API más limpio.
6      {
7      }
8  }
```

Bloque de código A.142: Prefiera el uso de objetos para tener APIs más limpios.

Volver a descripción de la regla.

```
1  public class MaybeAUtility {
2    public static void foo() {}
3    public static void bar() {}
```

```
}
```

Bloque de código A.143: Use clases utilitarias para aquellas que sólo tienen métodos estáticos.

A.1.4. Ejemplos de reglas de documentación Java.

Volver a descripción de la regla.

```
// OMG, this is horrible, Bob is an idiot !!!
```

Bloque de código A.144: Contenido de comentario.

Volver a descripción de la regla.

```
/**
* <Es requerido un comentario>
*
* @author Jon Doe
*/
```

Bloque de código A.145: Comentario requerido.

Volver a descripción de la regla.

```
/**
*
*   Comentario contiene demasiadas líneas.
*
*
*
*
*
*
*
*
*
*
*
*
```

```
16 */
```

Bloque de código A.146: Tamaño de comentario.

Volver a descripción de la regla.

```
1 public Foo() {
2   // Este constructor está vacío de forma intencional. Reporta la ausencia de
      este comentario.
3 }
```

Bloque de código A.147: Constructor vacío sin comentarios.

Volver a descripción de la regla.

```
1 public void doSomething() {
2   // Este método está vacío de forma intencional. Reporta la ausencia de este
      comentario.
3 }
```

Bloque de código A.148: Método vacío sin comentarios.

A.1.5. Ejemplos de reglas propensas a errores Java.

Volver a descripción de la regla.

```
1 public void bar() {
2     int x = 2;
3     if ((x = getX()) == 3) { // Incorrecto. evite asignaciones en operandos.
4       System.out.println("3!");
5     }
6 }
```

Bloque de código A.149: Evite la asignación en operandos.

Volver a descripción de la regla.

```
1 public class StaticField {
2    static int x;
3    public FinalFields(int y) {
4     x = y; // Inseguro, posible lectura o escritura sucia.
```

```
5     }
6 }
```

Bloque de código A.150: Asignación a campos `static` de varibles no `final`.

Volver a descripción de la regla.

```
1  import java.lang.reflect.Constructor;
2  import java.lang.reflect.Field;
3  import java.lang.reflect.Method;
4  import java.security.AccessController;
5  import java.security.PrivilegedAction;
6  public class Violation {
7      private void invalidSetAccessCalls() throws NoSuchMethodException,
           SecurityException {
8          Constructor<?> constructor = this.getClass().getDeclaredConstructor(
               String.class);
9          // Evite llamados a setAccessible
10         constructor.setAccessible(true);
11         Method privateMethod = this.getClass().getDeclaredMethod("
               aPrivateMethod");
12         // Evite llamados a setAccessible
13         privateMethod.setAccessible(true);
14         // Alteración de accesibilidad.
15         String privateField = AccessController.doPrivileged(new
               PrivilegedAction<String>() {
16             @Override
17             public String run() {
18                 try {
19                     Field field = Violation.class.getDeclaredField("
                           aPrivateField");
20                     field.setAccessible(true);
21                     return (String) field.get(null);
22                 } catch (ReflectiveOperationException | SecurityException e) {
23                     throw new RuntimeException(e);
24                 }
25             }
```

```
26          });
27      }
28  }
```

Bloque de código A.151: Evite la alteración de accesibilidad.

Volver a descripción de la regla.

```
1  public class A {
2      public class Foo {
3          String assert = "foo"; // assert es una palabra reservada desde java
               1.4
4      }
5  }
```

Bloque de código A.152: Evite el uso de la palabra `assert` como identificador.

Volver a descripción de la regla.

```
1  // Bifurcación inusual en el ciclo, evite este comportamiento.
2  for (int i = 0; i < 10; i++) {
3      if (i*i <= 25) {
4          continue;
5      }
6      break;
7  }
8  // Use la bifurcación del condicional como camino de salida.
9  for (int i = 0; i < 10; i++) {
10     if (i*i > 25) {
11         break;
12     }
13 }
```

Bloque de código A.153: Evite bifurcaciones inusules para interrumpuir la ejecución de un ciclo..

Volver a descripción de la regla.

```
1  // Bifurcación inusual en el ciclo, evite este comportamiento.
2  void foo() {
3      Bar b = new Bar();
4      b.finalize(); // Evite el llamado a finalize.
```

```
5 }
```

Bloque de código A.154: Evite el llamado explícito al método `finalize`.

Volver a descripción de la regla.

```
1 public class Foo {
2     void bar() {
3         try {
4         } catch (NullPointerException npe) { // No capture
            NullPointerException
5         }
6     }
7 }
```

Bloque de código A.155: Evite capturar `NullPointerException`.

Volver a descripción de la regla.

```
1 public class Foo {
2     void bar() {
3         try {
4         } catch (Throwable th) {  // No capture Throwable
5             th.printStackTrace();
6         }
7     }
8 }
```

Bloque de código A.156: Evite capturar `Throwable`.

Volver a descripción de la regla.

```
1 BigDecimal bd = new BigDecimal(1.123);       // Pierde precisión.
2 BigDecimal bd = new BigDecimal("1.123");     // Use cadenas para precisión
    exacta.
3 BigDecimal bd = new BigDecimal(12);          // No es necesario para números
    enteros.
```

Bloque de código A.157: Evite literales decimales en el constructor de `BigDecimal`.

Volver a descripción de la regla.

```
private void bar() {
     buz("Howdy"); // Sentencia duplicada.
     buz("Howdy");
     buz("Howdy");
     buz("Howdy");
}
private void buz(String x) {}
```

Bloque de código A.158: Evite literales duplicados.

Volver a descripción de la regla.

```
public class A {
    public class Foo {
        String enum = "foo"; // enum es una palabra reservada desde java 1.5
    }
}
```

Bloque de código A.159: Evite el uso `enum` como identificador.

Volver a descripción de la regla.

```
public class Foo {
    Object bar;
    // No se sabe si bar es un atributo o un método.
    void bar() {
    }
}
```

Bloque de código A.160: Evite usar la misma palabra para un atributo y método.

Volver a descripción de la regla.

```
public class Foo extends Bar {
    int foo;    // El atributo tiene el mismo nombre de la clase.
}
public interface Operation {
    int OPERATION = 1; // El atributo tiene el mismo nombre de la clase.
}
```

Bloque de código A.161: Evite usar la misma palabra para un atributo y su misma clase.

Volver a descripción de la regla.

```
try {
} catch (Exception ee) {
    // Evite este condicional.
    if (ee instanceof IOException) {
        cleanup();
    }
}
try {
} catch (IOException ee) {
    // Capture IOException directamente.
    cleanup();
}
```

Bloque de código A.162: Evite el uso de `instanceof` para validar el tipo de una excepción.

Volver a descripción de la regla.

```
private static final int MAX_NUMBER_OF_REQUESTS = 10;
public void checkRequests() {
    if (i == 10) {                        // Incorrecto.
      doSomething();
    }
    if (i == MAX_NUMBER_OF_REQUESTS) {    // Correcto
      doSomething();
    }
    if (aString.indexOf('.') != -1) {}  // No recomendado.
    if (aString.indexOf('.') >= 0) { }  // Correcto.
    if (aDouble > 0.0) {}               // Incorrecto.
    if (aDouble >= Double.MIN_VALUE) {} // Correcto.
    if (i == pos + 5) {}                // No recomendado
    if (i == pos + SUFFIX_LENGTH) {}    // Alternativa preferida.
}
```

Bloque de código A.163: Evite el uso de literales en condicionales.

Volver a descripción de la regla.

```
public void bar() {
```

```
2     try {
3     } catch (SomeException se) {
4         se.getMessage(); // se pierde información de la excepción.
5     }
6 }
```

Bloque de código A.164: No pierda información de la excepción.

Volver a descripción de la regla.

```
1  // Múltiples operadores unarios crean confusión.
2  int i = - -1;
3  int j = + - +1;
4  int z = ~~2;
5  boolean b = !!true;
6  boolean c = !!!true;
7  // Equivalentes al bloque anterior.
8  int i = 1;
9  int j = -1;
10 int z = 2;
11 boolean b = true;
12 boolean c = false;
```

Bloque de código A.165: Evite el uso de múltiples operadores unarios.

Volver a descripción de la regla.

```
1 nt i = 012;     // Asigna el valor de 10 no 12
2 int j = 010;    // Asigna el valor de 8 no 10
3 k = i * j;     // Asigna el valor de 80 no 120
```

Bloque de código A.166: Evite el uso de valores octales.

Volver a descripción de la regla.

```
1 public String bar(String string) {
2   // Debe usarse && para evitar un NullPointerException
3     if (string!=null || !string.equals(""))
4         return string;
5   // Debe ser || para evaluar ambos casos.
6     if (string==null && string.equals(""))
```

```
7         return string;
8 }
```

Bloque de código A.167: Validaciones de `null` incorrectas.

Volver a descripción de la regla.

```
1 public class DummyActivity extends Activity {
2     public void onCreate(Bundle bundle) {
3         // llamada a super.onCreate(bundle) faltante.
4         foo();
5     }
6 }
```

Bloque de código A.168: `super` debe llamarse primero.

Volver a descripción de la regla.

```
1 public class DummyActivity extends Activity {
2     public void onPause() {
3         foo();
4         // Llamada a super.onPause() está faltante.
5     }
6 }
```

Bloque de código A.169: `super` debe llamarse al final.

Volver a descripción de la regla.

```
1 public class Foo {
2    private FileInputStream _s = new FileInputStream("file");
3    public void skip(int n) throws IOException {
4       _s.skip(n); // No se sabe exactamente cuantos bytes se saltaron
          efectivamente.
5    }
6    public void skipExactly(int n) throws IOException {
7       while (n != 0) {// validación de bytes saltados
8          long skipped = _s.skip(n);
9          if (skipped == 0)
10            throw new EOFException();
11         n -= skipped;
```

```
12        }
13    }
```

Bloque de código A.170: Valide el resultado de `skip();` para `FileInputStream`.

Volver a descripción de la regla.

```
1 Collection c = new ArrayList();
2 Integer obj = new Integer(1);
3 c.add(obj);
4 // Esta conversión lanzaría una excepción de tipo ClassCastException.
5 Integer[] a = (Integer [])c.toArray();
6 // Correcto.
7 Integer[] b = (Integer [])c.toArray(new Integer[c.size()]);
```

Bloque de código A.171: Evite la generación de `ClassCastException` al utilizar `toArray`.

Volver a descripción de la regla.

```
1 public class Foo implements Cloneable {
2     @Override
3     protected Object clone() throws CloneNotSupportedException { // Incorrecto
        . método clone debe ser público
4     }
5 }
6 public class Foo implements Cloneable {
7     @Override
8     public Object clone() // Correcto.
9 }
```

Bloque de código A.172: Método `clone` debe ser público.

Volver a descripción de la regla.

```
1 public class MyClass { // Clase debe implementar Cloneable.
2  public Object clone() throws CloneNotSupportedException {
3   return foo;
4  }
5 }
```

Bloque de código A.173: Método `clone` debe implementar `Cloneable`.

Volver a descripción de la regla.

```
public class Foo implements Cloneable {
    @Override
    protected Object clone() { // Incorrecto. Método debe retornar Foo.
    }
}
public class Foo implements Cloneable {
    @Override
    public Foo clone() { // Correcto.
    }
}
```

Bloque de código A.174: Tipo de retorno de método `clone` debe retornar la clase que lo implementa.

Volver a descripción de la regla.

```
public class Bar {
    public void withSQL() {
        Connection c = pool.getConnection();
        try {
        } catch (SQLException ex) {
        } finally {
            // Incorrecto. La conexión c debe cerrarse con 'c.close();'
        }
    }
    public void withFile() {
        InputStream file = new FileInputStream(new File("/tmp/foo"));
        try {
            int c = file.in();
        } catch (IOException e) {
        } finally {
            // Incorrecto. El archivo 'file' debe cerrarse con 'file.close();'
        }
    }
}
```

Bloque de código A.175: Cierre recursos.

Volver a descripción de la regla.

```
class Foo {
  boolean bar(String a, String b) {
    return a == b; // Incorrecto.
    return a.equals(b); // Correcto.
  }
}
```

Bloque de código A.176: Evite el uso de == para comparar igualdad entre objetos.

Volver a descripción de la regla.

```
boolean x = (y == Double.NaN);  // Incorrecto.
boolean x = Double.isNaN(y);   // Correcto.
```

Bloque de código A.177: Compare correctamente con “*NaN*”.

Volver a descripción de la regla.

```
public class SeniorClass {
  public SeniorClass(){
      toString(); // Puede lanzar una NullPointerException si el método es
         sobreescrito
  }
  public String toString(){
    return "IAmSeniorClass";
  }
}
public class JuniorClass extends SeniorClass {
  private String name;
  public JuniorClass(){
    super(); // Automáticamente, lanza un NullPointerException.
    name = "JuniorClass";
  }
  public String toString(){
    return name.toUpperCase();
  }
```

```
18 }
```

Bloque de código A.178: Evite llamar métodos que se pueden sobre-escribir en el Constructor de una clase.

Volver a descripción de la regla.

```
1 public class MyTest {
2     @Test
3     public void someTest() {
4     }
5     // Incorrecto. Test no anotado
6     public void someOtherTest () {
7     }
8 }
```

Bloque de código A.179: Test no anotado.

Volver a descripción de la regla.

```
1 public class GCCall {
2     public GCCall() {
3         // Incorrecto. No llame al recolector de basura.
4         System.gc();
5     }
6     public void doSomething() {
7         // Incorrecto. No llame al recolector de basura.
8         Runtime.getRuntime().gc();
9     }
10     public explicitGCcall() {
11         // Incorrecto. No llame al recolector de basura.
12         System.gc();
13     }
14     public void doSomething() {
15         // Incorrecto. No llame al recolector de basura.
16         Runtime.getRuntime().gc();
17     }
18 }
```

Bloque de código A.180: No llame al recolector de basura de forma explícita.

Volver a descripción de la regla.

```
1 public class Foo extends Throwable { } // Incorrecto.
```

Bloque de código A.181: No extienda `Throwable`.

Volver a descripción de la regla.

```
1 public class MyActivity extends Activity {
2     protected void foo() {
3         String storageLocation = "/sdcard/mypackage";   // Incorrecto. no
              utilice sdcard
4        storageLocation = Environment.getExternalStorageDirectory() + "/
             mypackage"; // Correcto.
5     }
6 }
```

Bloque de código A.182: No extienda `Throwable`.

Volver a descripción de la regla.

```
1 public void bar() {
2     System.exit(0);                    // Nunca llame a System.exit
3 }
4 public void foo() {
5     Runtime.getRuntime().exit(0);   // Nunca pare la maquina virtual de Java
         manualmente, el contenedor de aplicaciones lo hace automáticamente.
6 }
```

Bloque de código A.183: No termine la maquina virutal de Java.

Volver a descripción de la regla.

```
1 public class Foo {
2     public void bar() {
3         try {
4         } catch( Exception e) {
5         } finally {
6             // Incorrecto. no lance excepciones en el bloque finally.
7             throw new Exception();
8         }
```

```
    }
}
```

Bloque de código A.184: No lance excepciones en el bloque `finally`.

Volver a descripción de la regla.

```
import sun.misc.foo; // Incorrecto.
public class Foo {}
```

Bloque de código A.185: No llame a los paquetes "*Sun*".

Volver a descripción de la regla.

```
public class Count {
  public static void main(String[] args) {
    final int START = 2000000000;
    int count = 0;
    for (float f = START; f < START + 50; f++) { // Incorrecto. el
       comportamiento del ciclo puede ser inesperado.
      count++;
      System.out.println(count);
    }
}
```

Bloque de código A.186: No use flotantes como índices en ciclos `for`.

Volver a descripción de la regla.

```
public void doSomething() {
    try {
        FileInputStream fis = new FileInputStream("/tmp/bugger");
    } catch (IOException ioe) {
        // Incorrecto. alguna lógica debe ejecutarse.
    }
}
```

Bloque de código A.187: Evite bloques `catch` vacíos.

Volver a descripción de la regla.

```
public class Foo {
```

```
2   protected void finalize() {} // Incorrecto.
3 }
```

Bloque de código A.188: Evite métodos `finalizer()` vacíos.

Volver a descripción de la regla.

```
1 String x = "foo";
2 if (x.equals(null)) {   // si x es null puede retornar un NullPointerException
    .
3     doSomething();
4 }
5 if (x == null) {        // Preferido.
6     doSomething();
7 }
```

Bloque de código A.189: Evite usar `equals()` para comparar con `null`.

Volver a descripción de la regla.

```
1 protected void finalize() {
2     something();
3     // Debe llamarse el método super.finalize()
4 }
```

Bloque de código A.190: Método `finalize()` no llama a `super.finalize()`.

Volver a descripción de la regla.

```
1 protected void finalize() {
2     // No es necesario sobreescribir finalize si sólo se llama al método padre
        .
3     super.finalize();
4 }
```

Bloque de código A.191: Método `finalize()` sólo llama a `super.finalize()`.

Volver a descripción de la regla.

```
1 public class Foo {
2     protected void finalize(int a) {
3     }
```

```
}
```

Bloque de código A.192: Método `finalize()` sobrecargado.

Volver a descripción de la regla.

```
public void finalize() { // Incorrecto. El método finalize debe ser protected.
}
```

Bloque de código A.193: Método `finalize()` debe ser de visibilidad `protected`.

Volver a descripción de la regla.

```
public class Foo {
 public void bar() {
  int x = 2;
  x = x; // operación idempotente.
 }
}
```

Bloque de código A.194: Evite operaciones idempotentes.

Volver a descripción de la regla.

```
public void bar(int status) {
    switch(status) {
      case CANCELLED:
        doCancelled();
        // Incorrecto. falta la sentencia break;
      case NEW:
        doNew();
        // Incorrecto. falta la sentencia break;
      case REMOVED:
        doRemoved();
        // Incorrecto. falta la sentencia break;
      case OTHER:
        // Incorrecto. Caso que siempre se ejecutaría.
      case ERROR:
        doErrorHandling();
        break;
    }
```

```
18 }
```

Bloque de código A.195: Agregue `break` en todos los casos de un condicional `switch`.

Volver a descripción de la regla.

```
1 // Incorrecto. No es necesario crear una instancia de String
2 Class c = new String().getClass();
3 // Correcto.
4 Class c = String.class;
```

Bloque de código A.196: Evite la instanciación de un objeto sólo para obtener su clase.

Volver a descripción de la regla.

```
1 LOGGER.error("forget the arg {}"); // Incorrecto. falta un parámetro.
2 LOGGER.error("forget the arg %s"); // Incorrecto. falta un parámetro.
3 LOGGER.error("too many args {}", "arg1", "arg2"); // Incorrecto. tiene un
     parámetro extra.
4 LOGGER.error("param {}", "arg1", new IllegalStateException("arg")); //
     Correcto. la excepción se muestra en una línea separada.
```

Bloque de código A.197: Formato de mensaje de `log` incorrecto.

Volver a descripción de la regla.

```
1 public class JumbledIncrementerRule1 {
2     public void foo() {
3         for (int i = 0; i < 10; i++) {          // Correcto. sólo referencia '
             i'
4             for (int k = 0; k < 20; i++) {      // Incorrecto. referencia 'i'
                 y 'k'
5                 System.out.println("Hello");
6             }
7         }
8     }
9 }
```

Bloque de código A.198: Evite desorden en el incremento de variables de control en `for` anidados.

Volver a descripción de la regla.

```
1 import junit.framework.*;
2 public class Foo extends TestCase {
3     public void setup() {}    // Incorrecto. debe ser setUp
4     public void TearDown() {} // Incorrecto. debe ser tearDown
5 }
```

Bloque de código A.199: Uso incorrecto de mayúsculas y minúsculas en métodos fijos de JUnit.

Volver a descripción de la regla.

```
1 import junit.framework.*;
2 public class Foo extends TestCase {
3     public void suite() {}         // Incorrecto. Debe ser static
4     private static void suite() {} // Incorrecto. Debe ser public
5 }
```

Bloque de código A.200: Convención de método `suite()` para test de JUnit.

Volver a descripción de la regla.

```
1 public class MyClass {
2     public MyClass() {}        // Correcto. Es el constructor de la clase
3     public void MyClass() {}    // Incorrecto. Los métodos no pueden llamarse
         igual que su clase.
4 }
```

Bloque de código A.201: Métodos no pueden tener el mismo nombre de su clase.

Volver a descripción de la regla.

```
1 public class Foo {
2     void bar() {
3         if (a.equals(baz) || a == null) {} // Incorrecto. Si a es null hay un
             NullPointerException.
4         if (a == null || a.equals(baz)) {} // Correcto.
5     }
6 }
```

Bloque de código A.202: Comparación con `null` mál ubicada en condicional.

Volver a descripción de la regla.

```
1 public class Foo implements java.io.Serializable {
2     // Si la clase implementa serializable de tener su serial version UID
3     // ejemplo: public static final long serialVersionUID = 4328743;
4     String name;
5 }
```

Bloque de código A.203: Ausencia del serial de versión “*UID*”.

Volver a descripción de la regla.

```
1 // Esta clase no se puede usar porque su constructor es privado y no tiene
    ningún método estático.
2 public class Foo {
3   private Foo() {}
4   void foo() {}
5 }
```

Bloque de código A.204: Método `static` ausente en clase con constructor privado.

Volver a descripción de la regla.

```
1 public class Foo {
2     Logger log = Logger.getLogger(Foo.class.getName());
3     // No es necesario tener 2 logger en una misma clase.
4     Logger log2= Logger.getLogger(Foo.class.getName());
5 }
```

Bloque de código A.205: Más de un `LOGGER`.

Volver a descripción de la regla.

```
1 public class Foo {
2   void bar(int a) {
3    switch (a) {
4      case 1:
5        break;
6      mylabel: // Compila, pero es confuso.
7        break;
8      default:
9        break;
10    }
```

```
11    }
12 }
```

Bloque de código A.206: Evite etiquetas en condicionales `switch`.

Volver a descripción de la regla.

```
1  class Buzz implements java.io.Serializable {
2      private static final long serialVersionUID = 1L;
3      private transient int someFoo;           // Correcto.
4      private static int otherFoo;             // Correcto.
5      private java.io.FileInputStream stream; // Incorrecto. FileInputStream no
           es serializable
6      public void setStream(FileInputStream stream) {
7          this.stream = stream;
8      }
9      public int getSomeFoo() {
10           return this.someFoo;
11     }
12 }
```

Bloque de código A.207: Clase no serializable.

Volver a descripción de la regla.

```
1  public class MyClass {
2    // Este bloque corre antes de cualquier llamada o inicialización de
         constructor, puede generar defectos difíciles de encontrar.
3    {
4      System.out.println("I am about to construct myself");
5    }
6  }
```

Bloque de código A.208: Inicializador no estático.

Volver a descripción de la regla.

```
1  public void bar() {
2    Object x = null; // Correcto.
3    x = new Object(); // Correcto.
```

```
4    x = null; // Incorrecto. si x no sufrió transformaciones no es necesario
         asignar null.
5  }
```

Bloque de código A.209: Asignación de `null` innecesaria.

Volver a descripción de la regla.

```
1  public class Bar {         // Incorrecto. falta el método hashcode()
2      public boolean equals(Object o) {
3      }
4  }
5  public class Baz {         // Incorrecto. falta el método equals()
6      public int hashCode() {
7      }
8  }
9  public class Foo {         // Correcto
10     public boolean equals(Object other) {
11     }
12     public int hashCode() {
13     }
```

Bloque de código A.210: Sobreescriba `equals` y `hashCode`.

Volver a descripción de la regla.

```
1  class Foo{
2      public Object clone(){
3          return new Foo(); // Incorrecto. retornar una nueva instancia no es
               una forma apropiada de clonar el objeto.
4      }
5  }
```

Bloque de código A.211: Mala implementación de `clone`.

Volver a descripción de la regla.

```
1  public class Foo {
2      private static final Log LOG = LogFactory.getLog(Foo.class);    //
           Correcto.
```

```
3     protected Log LOG = LogFactory.getLog(Testclass.class);          //
          Incorrecto.
4 }
```

Bloque de código A.212: Logger debe ser privado y `final`.

Volver a descripción de la regla.

```
1  public class Example {
2      // Incorrecto
3      public int[] badBehavior() {
4          return null;
5      }
6      // Correcto.
7      public String[] bonnePratique() {
8          return new String[0];
9      }
10 }
```

Bloque de código A.213: Retorne una colección vacía en vez de `null`.

Volver a descripción de la regla.

```
1  public class Bar {
2      public String foo() {
3          try {
4              throw new Exception( "My Exception" );
5          } catch (Exception e) {
6              throw e;
7          } finally {
8              return "A. O. K."; // Incorrecto.
9          }
10     }
11 }
```

Bloque de código A.214: No retorne en el bloque `finally`.

Volver a descripción de la regla.

```
1  public class Foo {
2    // Debe especificar el Locale.US o el necesario.
```

```
3   private SimpleDateFormat sdf = new SimpleDateFormat("pattern");
4 }
```

Bloque de código A.215: `SimpleDateFormat` necesia de un `Locale`.

Volver a descripción de la regla.

```
1  public class Singleton {
2      private static Singleton singleton = new Singleton( );
3      private Singleton(){ }
4      public static Singleton getInstance() {
5          return singleton;
6      }
7      public static Singleton getInstance(Object obj){
8          Singleton singleton = (Singleton) obj;
9          return singleton;           //Incorrecto.
10     }
11 }
```

Bloque de código A.216: `Singleton` con un único método.

Volver a descripción de la regla.

```
1 class Singleton {
2     private static Singleton instance = null;
3     public static Singleton getInstance() {
4         synchronized(Singleton.class) {
5             return new Singleton(); // Incorrecto debe asignarse al campo
                  instance.
6         }
7     }
8 }
```

Bloque de código A.217: `Singleton` retorna una nueva instancia.

Volver a descripción de la regla.

```
1 public class SomeEJB extends EJBObject implements EJBLocalHome {
2     private static int CountA;          // Incorrecto.
3     private static final int CountB;    // Correcto.
```

```
4 }
```

Bloque de código A.218: EJB estático debe ser `final`.

Volver a descripción de la regla.

```
1 new StringBuilder()     //  16
2 new StringBuilder(6)    //  6
3 new StringBuilder("hello world")  // 11 + 16 = 27
4 new StringBuilder('')   //  chr(C) = 67
5 new StringBuffer('A')   // Incorrecto.  chr(A) = 65
6 new StringBuilder("A")   //  Correcto. 1 + 16 = 17
```

Bloque de código A.219: Nueva instancia de `StringBuffer/StringBuilder` con una caracter.

Volver a descripción de la regla.

```
1  public class Foo {
2    public int equals(Object o) {
3      // Incorrecto. probablemente retorna un boolean.
4    }
5    public boolean equals(String s) {
6      // Incorrecto. probablemente es equals(Object)
7    }
8    public boolean equals(Object o1, Object o2) {
9      // Incorrecto. probablemente es equals(Object)
10   }
11 }
```

Bloque de código A.220: Método `equals` sospechoso.

Volver a descripción de la regla.

```
1 public class Foo {
2     public int hashcode() { // Incorrecto. probablemente es 'hashCode'
3     }
4 }
```

Bloque de código A.221: Método `hashCode` sospechoso.

Volver a descripción de la regla.

```
1 public void foo() {
2   // Incorrecto se interpreta como 12, seguido del caracter '8'
3   System.out.println("suspicious: \128");
4 }
```

Bloque de código A.222: Escape Octal sospechoso.

Volver a descripción de la regla.

```
1 // Si la clase no tiene test, considere renombrar la clase.
2 public class CarTest {
3    public static void main(String[] args) {
4    }
5 }
```

Bloque de código A.223: Clase de test sin casos.

Volver a descripción de la regla.

```
1 public class Foo {
2     public void close() {
3         if (true) { // Incorrecto.
4         }
5     }
6 }
```

Bloque de código A.224: Condicional `if` sin condición valida.

Volver a descripción de la regla.

```
1 public class SimpleTest extends TestCase {
2     public void testX() {
3         assertTrue(true);       // No tiene un propósito válido.
4     }
5 }
```

Bloque de código A.225: Aserción booleana innecesaria.

Volver a descripción de la regla.

```
1 boolean answer1 = buz.toUpperCase().equals("baz"); // Puede ser reemplazado
    por: buz.equalsIgnoreCase("baz")
```

Bloque de código A.226: Conversión de mayúsculas/minúsculas innecesar.

Volver a descripción de la regla.

```
1 public String convert(int x) {
2     String foo = new Integer(x).toString(); // Desperdicia una variable
3     return Integer.toString(x);             // Correcto.
4 }
```

Bloque de código A.227: Conversión temporal innecesaria.

Volver a descripción de la regla.

```
1  public class Test {
2      public String method1() { return "ok";}
3      public String method2() { return null;}
4      public void method(String a) {
5          String b;
6          if (a!=null && method1().equals(a)) { // Incorrecto
7          }
8          if (method1().equals(a) && a != null) {
9          }
10         if (a!=null && method1().equals(b)) {
11         }
12         if (a!=null && "LITERAL".equals(a)) {
13         }
14         if (a!=null && !a.equals("go")) {
15             a=method2();
16             if (method1().equals(a)) {
17             }
18         }
19     }
20 }
```

Bloque de código A.228: Comparación con `null` innecesaria.

Volver a descripción de la regla.

```
1 public class Main {
2     private static final Log _LOG = LogFactory.getLog( Main.class );
3     void bar() {
4         try {
```

```
5         } catch( Exception e ) {
6             _LOG.error( e ); //Incorrecto.
7         } catch( OtherException oe ) {
8             _LOG.error( oe.getMessage(), oe ); //Correcto.
9         }
10     }
11 }
```

Bloque de código A.229: Registre excepciones de forma correcta.

Volver a descripción de la regla.

```
1 public boolean test(String s) {
2     if (s == "one") return true;         // no confiable.
3     if ("two".equals(s)) return true;    // Correcto.
4     return false;
5 }
```

Bloque de código A.230: Use `equals` para comparar `String`.

Volver a descripción de la regla.

```
1 import java.math.*;
2 class Test {
3     void method1() {
4         BigDecimal bd=new BigDecimal(10);
5         bd.add(new BigDecimal(5));       // Incorrecto.
6     }
7     void method2() {
8         BigDecimal bd=new BigDecimal(10);
9         bd = bd.add(new BigDecimal(5)); // Correcto.
10     }
11 }
```

Bloque de código A.231: Operación innecesaria en inmutable.

Volver a descripción de la regla.

```
1 // Incorrecto.
2 if (x.toLowerCase().equals("list")) {}
3 // Correcto.
```

```
4 if (x.toLowerCase(Locale.US).equals("list")) { }
5 // o use
6 if (x.equalsIgnoreCase("list")) { }
7 // Correcto.
8 String z = a.toLowerCase(Locale.ROOT);
9 // Correcto.
10 String z2 = a.toLowerCase(Locale.getDefault());
```

Bloque de código A.232: Convierta a mayúsculas/minúsculas usando `Locale`.

Volver a descripción de la regla.

```
1 public class Foo {
2     ClassLoader cl = Bar.class.getClassLoader(); // Incorrecto.
3     ClassLoader cl = Thread.currentThread().getContextClassLoader() //
          Correcto
4 }
```

Bloque de código A.233: Use el `ClassLoader` correcto.

A.1.6. Ejemplos de reglas de multihilo Java.

Volver a descripción de la regla.

```
1 public class Foo {
2     // Incorrecto
3     synchronized void foo() {
4         // sólo este condicional requiere sincronización
5         if (!sharedData.has("bar")) {
6             sharedData.add("bar");
7         }
8     }
9     // Correcto
10    void bar() {
11        synchronized(this) {
12            if (!sharedData.has("bar")) {
13                sharedData.add("bar");
14            }
```

```
15          }
16      }
17      // Evite sincronización para métodos estáticos
18      static synchronized void fooStatic() {
19      }
20      // Correcto
21      static void barStatic() {
22          synchronized(Foo.class) {
23          }
24          // más código que no necesita sincronización.
25      }
26  }
```

Bloque de código A.234: Evite el uso de `syncronized` a nivel de método.

Volver a descripción de la regla.

```
1  public class Bar {
2      void buz() {
3          ThreadGroup tg = new ThreadGroup("My threadgroup"); // Incorrecto.
4          tg = new ThreadGroup(tg, "my thread group");
5          tg = Thread.currentThread().getThreadGroup();
6          tg = System.getSecurityManager().getThreadGroup();
7      }
8  }
```

Bloque de código A.235: Evite el uso de `AvoidThreadGroup`.

Volver a descripción de la regla.

```
1  public class ThrDeux {
2    private volatile String var1; // Incorrecto.
3    private          String var2; // Correcto.
4  }
```

Bloque de código A.236: Evite el uso de `volatile`.

Volver a descripción de la regla.

```
1  // Evite el uso de Thread
2  public class UsingThread extends Thread {
```

```
3  }
4  // Evite el uso de ExecutorService
5  public class UsingExecutorService {
6      public void methodX() {
7          ExecutorService executorService = Executors.newFixedThreadPool(5);
8      }
9  }
10 // Evite el uso de ExecutorService
11 public class Example implements ExecutorService {
12 }
13 // Evite el uso de AbstractExecutorService
14 public class Example extends AbstractExecutorService {
15 }
16 // Evite el uso de Executors
17 public class UsingExecutors {
18     public void methodX() {
19         Executors.newSingleThreadExecutor().submit(() -> System.out.println("
               Hello!"));
20     }
```

Bloque de código A.237: Evite el uso de `Thread`.

Volver a descripción de la regla.

```
1 Thread t = new Thread();
2 t.run();            // Incorrecto. use t.start()
3 new Thread().run(); // Incorrecto. use Thread.start()
```

Bloque de código A.238: No llame `Thread.run()`.

Volver a descripción de la regla.

```
1 public class Foo {
2     /* volatile */ Object baz = null; // Para java 5 se debe usar volatile
3     Object bar() {
4         if (baz == null) { // Podría ser no nula pero no aún no se ha creado
              totalmente.
5             synchronized(this) {
6                 if (baz == null) {
```

```
                    baz = new Object();
                }
            }
        }
        return baz;
    }
}
```

Bloque de código A.239: Use volatile para bloques sincronizados.

Volver a descripción de la regla.

```
private static Foo foo = null;
// Llamados simultáneos pueden leer objetos parcialmente inicializados
public static Foo getFoo() {
    if (foo==null) {
        foo = new Foo();
    }
    return foo;
}
```

Bloque de código A.240: Singleton declarado sin manejar hilo seguro.

Volver a descripción de la regla.

```
public class Foo {
    private static final SimpleDateFormat sdf = new SimpleDateFormat();
    void bar() {
        sdf.format(); // Incorrecto. no es hilo seguro.
    }
    void foo() {
        synchronized (sdf) { // Correcto. use un bloque sincronizado.
            sdf.format();
        }
    }
}
```

Bloque de código A.241: Formateador de fecha no sincronizado.

Volver a descripción de la regla.

```
1 public class ConcurrentApp {
2   public void getMyInstance() {
3     Map map1 = new HashMap();            // Uselo para un hilo simple.
4     Map map2 = new ConcurrentHashMap(); // Use para acceso de múltiples hilos.
5   }
6 }
```

Bloque de código A.242: Use `ConcurrentHashMap()`.

Volver a descripción de la regla.

```
1 void bar() {
2     x.notify(); // Incorrecto.
3     x.notifyAll(); // Correcto.
4   }
```

Bloque de código A.243: Use `notifyAll()` en vez de `notify()`.

A.1.7. Ejemplos de reglas de rendimiento de Java.

Volver a descripción de la regla.

```
1 String s = "" + 123;                // Incorrecto.
2 String t = Integer.toString(123);   // Correcto.
```

Bloque de código A.244: No sume un `String` vacío `""`.

Volver a descripción de la regla.

```
1 StringBuffer sb = new StringBuffer();
2 sb.append("a");      // Incorrecto.
3 sb.append('a');      // Correcto.
```

Bloque de código A.245: Use comillas sencillas para anexar un caracter.

Volver a descripción de la regla.

```
1 class Scratch {
2     void copy_a_to_b() {
3         int[] a = new int[10];
4         int[] b = new int[10];
```

```
5         // Incorrecto.
6         for (int i = 0; i < a.length; i++) {
7             b[i] = a[i];
8         }
9         // Correcto
10        b = Arrays.copyOf(a, a.length);
11        // Correcto
12        System.arraycopy(a, 0, b, 0, a.length);
13    }
14    void shift_left(int[] a) {
15        // Incorrecto.
16        for (int i = 0; i < a.length - 1; i++) {
17            a[i] = a[i + 1];
18        }
19        // Correcto.
20        System.arraycopy(a, 1, a, 0, a.length - 1);
21    }
22    void shift_right(int[] a) {
23        // Incorrecto.
24        for (int i = a.length - 1; i > 0; i--) {
25            a[i] = a[i - 1];
26        }
27        // Correcto.
28        System.arraycopy(a, 0, a, 1, a.length - 1);
29    }
30 }
```

Bloque de código A.246: Evite iterar sobre arreglos usando ciclos `for` cuando no es necesario.

Volver a descripción de la regla.

```
1 import java.time.LocalDateTime;
2 import java.util.Calendar;
3 import java.util.Date;
4 public class DateStuff {
5     private Date bad1() {
6         return Calendar.getInstance().getTime(); // Incorrecto.
```

```
7       }
8       private Date good1a() {
9           return new Date(); // Correcto. // Java 7 y anteriores.
10      }
11      private LocalDateTime good1b() {
12          return LocalDateTime.now(); // Correcto. Java 8+
13      }
14      private long bad2() {
15          return Calendar.getInstance().getTimeInMillis(); // Incorrecto
16      }
17      private long good2() {
18          return System.currentTimeMillis(); // Correcto.
19      }
20  }
```

Bloque de código A.247: Evite usar la clase `Calendar` para crear fechas.

Volver a descripción de la regla.

```
1       // Incorrecto. el uso de FileStreams causa pausas en el recolector de
           basura de Java
2       FileInputStream fis = new FileInputStream(fileName);
3       FileOutputStream fos = new FileOutputStream(fileName);
4       FileReader fr = new FileReader(fileName);
5       FileWriter fw = new FileWriter(fileName);
6       // Correcto.
7       try(InputStream is = Files.newInputStream(Paths.get(fileName))) {
8       }
9       try(OutputStream os = Files.newOutputStream(Paths.get(fileName))) {
10      }
11      try(BufferedReader br = Files.newBufferedReader(Paths.get(fileName),
           StandardCharsets.UTF_8)) {
12      }
13      try(BufferedWriter wr = Files.newBufferedWriter(Paths.get(fileName),
           StandardCharsets.UTF_8)) {
14      }
```

Bloque de código A.248: Evite usar la clase `FileStream`.

Volver a descripción de la regla.

```
1 public class Something {
2     public static void main( String as[] ) {
3         for (int i = 0; i < 10; i++) {
4             Foo f = new Foo(); // Evite su uso, es una operación costosa.
5         }
6     }
7 }
```

Bloque de código A.249: Evite la instanciación de objetos dentro de ciclos.

Volver a descripción de la regla.

```
1 BigInteger bi = new BigInteger(1);        // use BigInteger.ONE  en su lugar
2 BigInteger bi2 = new BigInteger("0");     // use BigInteger.ZERO en su lugar
3 BigInteger bi3 = new BigInteger(0.0);     // use BigInteger.ZERO en su lugar
4 BigInteger bi4;
5 bi4 = new BigInteger(0);                  // use BigInteger.ZERO en su lugar
```

Bloque de código A.250: Evite la instanciación constantes `BigInteger`.

Volver a descripción de la regla.

```
1 String foo = " ";
2 // Incorrecto
3 StringBuffer buf = new StringBuffer();
4 buf.append("Hello");
5 buf.append(foo);
6 buf.append("World");
7 // Correcto
8 StringBuffer buf = new StringBuffer();
9 buf.append("Hello").append(foo).append("World");
```

Bloque de código A.251: Anexos (`append`) consecutivos deben ir en la misma línea.

Volver a descripción de la regla.

```
1 StringBuilder buf = new StringBuilder();
2 buf.append("Hello").append(" ").append("World");     // Incorrecto
3 buf.append("Hello World");                           // Correcto
```

```
4 buf.append('h').append('e').append('l').append('l').append('o'); // Incorrecto
5 buf.append("hello");                                             // Correcto
6 buf.append(1).append('m');  // Incorrecto.
7 buf.append("1m");           // Correcto.
```

Bloque de código A.252: Evite anexos (`append`) de literales consecutivos.

Volver a descripción de la regla.

```
1 public void bar(String string) {
2     if (string != null &&
3         string.trim().length() > 0) { // El uso de trim para validación de un
              String vacío es ineficiente.
4     }
5 }
```

Bloque de código A.253: Chequeo de String vacío ineficiente.

Volver a descripción de la regla.

```
1 // Incorrecto. Dos objetos inmutables se están creando acá.
2 StringBuffer sb = new StringBuffer("tmp = "+System.getProperty("java.io.tmpdir
    "));
3 // Correcto.
4 StringBuffer sb = new StringBuffer("tmp = ");
5 sb.append(System.getProperty("java.io.tmpdir"));
```

Bloque de código A.254: Inicialización de `StringBuilder/StringBuffer` ineficiente.

Volver a descripción de la regla.

```
1 StringBuilder bad = new StringBuilder();
2 bad.append("This is a long string that will exceed the default 16 characters")
    ; // Incorrecto. el String de inicialización supera los 16 caracteres por
    defecto.
3 StringBuilder good = new StringBuilder(41);
4 good.append("This is a long string, which is pre-sized");// Correcto.
```

Bloque de código A.255: Declaración de `StringBuilder/StringBuffer` ineficiente.

Volver a descripción de la regla.

```
1 List<Foo> foos = getFoos();
2 // Correcto.
3 Foo[] fooArray = foos.toArray(new Foo[0]);
4 // Incorrecto. el arreglo debe ser inicializado en 0 y luego asignar el tamaño
     correcto.
5 Foo[] fooArray = foos.toArray(new Foo[foos.size()]);
```

Bloque de código A.256: Use de `toArray()` optimizable.

Volver a descripción de la regla.

```
1 public class C {
2     boolean b   = false; // No es necesario incializar con el valor por
         defecto.
3     byte by     = 0; // No es necesario incializar con el valor por defecto.
4     short s     = 0; // No es necesario incializar con el valor por defecto.
5     char c      = 0; // No es necesario incializar con el valor por defecto.
6     int i       = 0; // No es necesario incializar con el valor por defecto.
7     long l      = 0; // No es necesario incializar con el valor por defecto.
8     float f     = .0f; // No es necesario incializar con el valor por defecto.
9     doable d    = 0d; // No es necesario incializar con el valor por defecto.
10    Object o    = null; // No es necesario incializar con el valor por defecto
         .
11    MyClass mca[] = null; // No es necesario incializar con el valor por
         defecto.
12    int i1 = 0, ia1[] = null; // No es necesario incializar con el valor por
         defecto.
13    class Nested {
14        boolean b = false; // No es necesario incializar con el valor por
             defecto.
15    }
16 }
```

Bloque de código A.257: Inicializadores de variables redundantes.

Volver a descripción de la regla.

```
1 private String bar = new String("bar"); // Prefiera el uso de: String bar = "
```

```
    bar";
```

Bloque de código A.258: Evite la instanciación de la clase `String`.

Volver a descripción de la regla.

```
1 private String baz() {
2     String bar = "howdy";
3     return bar.toString(); // No es necesario, la variable bar ya es un String
        .
4 }
```

Bloque de código A.259: Evite llamar `toString` de un campo `String`.

Volver a descripción de la regla.

```
1 // Al menos un switch debe tener 3 casos.
2 public class Foo {
3     public void bar() {
4         switch (condition) {
5             case ONE:
6                 instruction;
7                 break;
8             default:
9                 break;
10        }
11    }
12 }
```

Bloque de código A.260: Muy pocas bifurcaciones para sentencias `switch`.

Volver a descripción de la regla.

```
1 import java.util.*;
2 public class SimpleTest extends TestCase {
3     public void testX() {
4     Collection c1 = new Vector(); // No es necesario usar Vector en un sólo
        hilo de ejecución.
5     Collection c2 = new ArrayList();
6     }
```

```
7 }
```

Bloque de código A.261: Use `ArrayList` en vez de `Vector`.

Volver a descripción de la regla.

```
1 public class Test {
2     public void foo(Integer[] ints) {
3         // No es necesario iterar, simplemente use Arrays.asList(ints)
4         List<Integer> l = new ArrayList<>(100);
5         for (int i = 0; i < ints.length; i++) {
6             l.add(ints[i]);
7         }
8     }
9 }
```

Bloque de código A.262: Use `Array.asList()` en vez de iterar sobre un arreglo.

Volver a descripción de la regla.

```
1 String s = "hello world";
2 // Incorrecto.
3 if (s.indexOf("d") {}
4 // Correcto.
5 if (s.indexOf('d') {}
```

Bloque de código A.263: Use comilla sencilla `''` en el método `indexOf` para buscar un `char`.

Volver a descripción de la regla.

```
1 import org.apache.commons.fileupload.FileItem;
2 public class FileStuff {
3    private String bad(FileItem fileItem) { // Incorrecto.
4         return fileItem.getString();
5    }
6    private InputStream good(FileItem fileItem) { // Correcto.
7         return fileItem.getInputStream();
8    }
9 }
```

Bloque de código A.264: No use `get` en la clase `FileItem`.

Volver a descripción de la regla.

```
1 import org.apache.commons.fileupload.FileItem;
2 public String convert(int i) {
3     String s;
4     s = "a" + String.valueOf(i);    // No requerido.
5     s = "a" + i;                    // Correcto.
6     return s;
7 }
```

Bloque de código A.265: `String.valueOf` innecesario.

Volver a descripción de la regla.

```
1 public class Foo {
2     String inefficientConcatenation() {
3         String result = "";
4         for (int i = 0; i < 10; i++) {
5             // Ineficiente.
6             result += getStringFromSomeWhere(i);
7         }
8         return result;
9     }
10    String efficientConcatenation() {
11        // Correcto
12        StringBuilder result = new StringBuilder();
13        for (int i = 0; i < 10; i++) {
14            result.append(getStringFromSomeWhere(i));
15        }
16        return result.toString();
17    }
18 }
```

Bloque de código A.266: Use `StringBuilder/StringBuffer` para concatenar `String`.

Volver a descripción de la regla.

```
1 StringBuffer sb = new StringBuffer();
2 if (sb.toString().equals("")) {}        // Ineficiente
```

```
3 if (sb.length() == 0) {}                  // Correcto.
```

Bloque de código A.267: Use `length` para validar si un `StringBuilder`/`StringBuffer` está vacío.

A.1.8. Ejemplos de reglas de seguridad de Java.

Volver a descripción de la regla.

```
1 public class Foo {
2     void good() {
3         SecretKeySpec secretKeySpec = new SecretKeySpec(Properties.getKey(), "
              AES"); // Correcto.
4     }
5     void bad() {
6         SecretKeySpec secretKeySpec = new SecretKeySpec("my secret here".
              getBytes(), "AES");// Incorrecto.
7     }
8 }
```

Bloque de código A.268: No agregue información sensible como literales.

Volver a descripción de la regla.

```
1 public class Foo {
2     void good() { // Correcto.
3         SecureRandom random = new SecureRandom();
4         byte iv[] = new byte[16];
5         random.nextBytes(bytes);
6     }
7
8     void bad() { // Incorrecto.
9         byte[] iv = new byte[] { 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
              00, 00, 00, 00, 00, };
10    }
11
12    void alsoBad() { // Incorrecto.
13        byte[] iv = "secret iv in here".getBytes();
14    }
```

```
15 }
```

Bloque de código A.269: No use vectores/arreglos fijos para inicializar llaves criptográficas.

A.2. Bloques de código reglas de JavaScript

A.2.1. Ejemplos de buenas prácticas de JavaScript

Volver a descripción de la regla.

```
1 // Incorrecto
2 with (object) {
3     property = 3; // No es claro si está en un objeto, en una ventana, es
         confuso.
4 }
```

Bloque de código A.270: JavaScript Evite el uso de la sentencia `with` (PMD, 2022).

Volver a descripción de la regla.

```
1 // Correcto
2 function foo() {
3     if (condition1) {
4         return true;
5     }
6     return false;
7 }
8 // Incorrecto
9 function bar() {
10     if (condition1) {
11         return;
12     }
13     return false;
14 }
```

Bloque de código A.271: JavaScript retorno consistente (PMD, 2022).

Volver a descripción de la regla.

```
function(arg) {
    // Incorrecto.
    notDeclaredVariable = 1;     // Esta sentencia crea una variable global y
        lanza la regla
    // Correcto
    var someVar = 1;             // Ejemplo de una variable local, Correcto
    // Correcto.
    window.otherGlobal = 2;      // Esto no lanza la regla, aunque sea una
        variable global porque está encapsulada en la ventana.
}
```

Bloque de código A.272: JavaScript variables globales (PMD, 2022).

Volver a descripción de la regla.

```
// Correcto
function foo() {
    var p = 'clean';
    function() {
        var obj = { dirty: 'dirty' };
        for (var p in obj) { // Usar 'var' aquí.
            obj[p] = obj[p];
        }
        return x;
    }();
    // 'p' no fue modificada por el alcance del for in y aún su valor es = '
        clean'.
}
// Incorrecto
function bar() {
    var p = 'clean';
    function() {
        var obj = { dirty: 'dirty' };
        for (p in obj) { // La palabra clave var falta en el for in.
            obj[p] = obj[p];
        }
        return x;
```

```
22      }();
23      // 'p' fue modificada con el valor de obj 'dirty' porque el modificador
           var faltó en el 'for in'.
24  }
```

Bloque de código A.273: JavaScript retorno consistente (PMD, 2022).

Volver a descripción de la regla.

```
1  // Correcto
2  parseInt("10", 10);
3  //Incorrecto
4  parseInt("010");    // Podría ser interpretado como 10 o 7 (con base de 7).
```

Bloque de código A.274: JavaScript uso de base en `parseInt` (PMD, 2022).

A.2.2. Ejemplos de estilo de código de JavaScript

Volver a descripción de la regla.

```
1  var x = 2;
2  // Incorrecto
3  if ((x = getX()) == 3) {
4      alert('3!');
5  }
6  function getX() {
7      return 3;
8  }
```

Bloque de código A.275: JavaScript evite la asignación en operados (PMD, 2022).

Volver a descripción de la regla.

```
1  // Correcto
2  for (var i = 0; i < 42; i++) {
3      foo();
4  }
5  // Incorrecto
6  for (var i = 0; i < 42; i++)
```

```
7       foo();
```

Bloque de código A.276: JavaScript: Ciclos `for` deben usar llaves (PMD, 2022).

Volver a descripción de la regla.

```
1  // Correcto
2  if (foo) {
3      x++;
4  } else {
5      y++;
6  }
7  // Incorrecto
8  if (foo)
9      x++;
10 else
11     y++;
```

Bloque de código A.277: JavaScript: Condicionales `if`/`else` deben usar llaves (PMD, 2022).

Volver a descripción de la regla.

```
1  // Correcto
2  if (foo) {
3      x++;
4  }
5  // Incorrecto
6  if (foo)
7      x++;
```

Bloque de código A.278: JavaScript: Condicionales `if` deben usar llaves (PMD, 2022).

Volver a descripción de la regla.

```
1  // Incorrecto:
2  if (x) {
3      return y;
4  } else {
5      return z;
6  }
7  // Correcto:
```

```
8  if (x) {
9      return y;
10 }
11 return z;
```

Bloque de código A.279: JavaScript: Evite el uso de `else return` (PMD, 2022).

Volver a descripción de la regla.

```
1 if (foo) {
2     // Correcto
3 }
4 if (bar) {
5     {
6         // Incorrecto
7     }
8 }
```

Bloque de código A.280: JavaScript: Bloque innecesario (PMD, 2022).

Volver a descripción de la regla.

```
1 var x = 1; // Correcto
2 var y = (1 + 1); // Correcto
3 var z = ((1 + 1)); // Incorrecto
```

Bloque de código A.281: JavaScript: Paréntesis innecesarios (PMD, 2022).

Volver a descripción de la regla.

```
1  // Correcto
2  function foo() {
3      return 1;
4  }
5  // Incorrecto
6  function bar() {
7      var x = 1;
8      return x;
9      x = 2; // Esta línea nunca se ejecutaría.
10 }
```

Bloque de código A.282: JavaScript: Código inalcanzable (PMD, 2022).

Volver a descripción de la regla.

```
// Correcto
while (true) {
    x++;
}
// Incorrecto
while (true)
    x++;
```

Bloque de código A.283: JavaScript: Ciclos `while` deben usar llaves (PMD, 2022).

A.2.3. Ejemplos de código propenso a errores de JavaScript

Volver a descripción de la regla.

```
function(arg) {
    var obj1 = { a : 1 };    // Correcto
    var arr1 = [ 1, 2 ];     // Correcto

    var obj2 = { a : 1, };  // Incorrecto, error de sintáxis en algunos
        navegadores
    var arr2 = [ 1, 2, ];    // Incorrecto, la logitud podría ser 2 o 3
        dependiendo del navegador
}
```

Bloque de código A.284: JavaScript: Evite la coma final (PMD, 2022).

Volver a descripción de la regla.

```
// Correcto
if (someVar === true) {
  ...
}
// Correcto
if (someVar !== 3) {
  ...
}
// Incorrecto
```

```
10 if (someVar == true) {
11   ...
12 }
13 // Incorrecto
14 if (someVar != 3) {
15   ...
16 }
```

Bloque de código A.285: JavaScript: Comparación de igualdad (PMD, 2022).

Volver a descripción de la regla.

```
1 var a = 9; // Correcto
2 var b = 999999999999999; // Correcto
3 var c = 999999999999999999999; // Incorrecto
4 var w = 1.12e-4; // Correcto
5 var x = 1.12; // Correcto
6 var y = 1.1234567890123; // Correcto
7 var z = 1.12345678901234567; // Incorrecto
```

Bloque de código A.286: JavaScript: Literal numério inexacto (PMD, 2022).