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

**Anexo Reporte de Evaluación de Solución Tecnológica – Portal Empresarial (Port Empresarial)**

**15/nov/2018**

**Contenido**

[1. Objetivo 3](#_Toc519840867)

[2. Introducción 3](#_Toc519840868)

[3. Alcance 4](#_Toc519840869)

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

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

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

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

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

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

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

[4.7. Resultado Integral 13](#_Toc519840877)

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

[5. Conclusiones 19](#_Toc519840879)

1. Objetivo

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

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

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

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

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

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

1. Introducción

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

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

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



Ilustración Propuesta de Valor

1. Alcance

La evaluación de soluciones tecnológicas considera los siguientes aspectos:

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

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

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

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

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

El **Sistema de Portal Empresarial** tiene por objetivo permitir a los patrones efectuar trámites de forma electrónica.

* 1. Tipo de Solicitud

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

* **Descripción de la Solicitud**

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

* 1. Evaluación de Código Fuente

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

* **Categoría**
  + ***Bugs***

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

* + **Vulnerabilidades de Seguridad**

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

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

* + **Optimizadores**

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

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

* **Severidades**

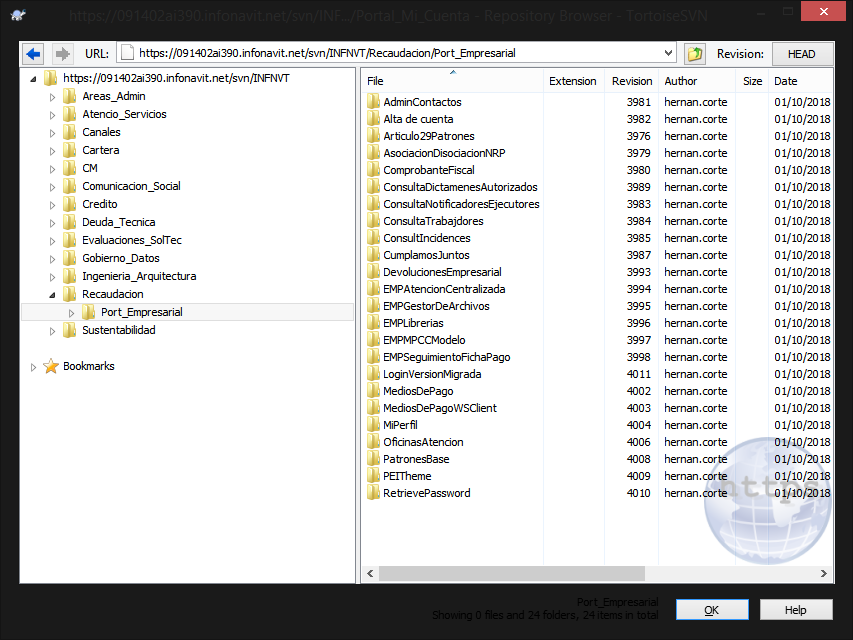
Los errores categorizados anteriormente son clasificados por severidades las cuales son definidas con base en 2 factores: **Impacto** y **Probabilidad**.

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

* + 1. Ubicación en el Repositorio Institucional

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

[*https://091402ai390.infonavit.net/svn/INFNVT/Recaudacion/Port\_Empresarial*](https://091402ai390.infonavit.net/svn/INFNVT/Recaudacion/Port_Empresarial)



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

AdminContactos

Alta\_de\_cuenta

Articulo29Patrones

AsociacionDisociacionNRP

ComprobanteFiscal

ConsultaDictamenesAutorizados

ConsultaNotificadoresEjecutores

ConsultaTrabajdores

ConsultIncidences

CumplamosJuntos

DevolucionesEmpresarial

EMPAtencionCentralizada

EMPGestorDeArchivos

EMPLibrerias

EMPMPCCModelo

EMPSeguimientoFichaPago

ListadoAplicativosPortalEmpresarial.xlsx

LoginVersionMigrada

MediosDePago

MediosDePagoWSClient

MiPerfil

OficinasAtencion

PatronesBase

PEITheme

RelacionAplicativosPedidoArquitectura.xlsx

RetrievePassword

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

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

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

# Caracteres Unicode con acentos:

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

#

# a - \u00e1 A - \u00c1

# e - \u00e9 E - \u00c9

# i - \u00ed I - \u00cd

# o - \u00f3 O - \u00d3

# u - \u00fa U - \u00da

#

# Caracteres Unicode con diéresis

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

# u - \u00fc U - \u00dc

#

# Caracteres Unicode con tilde

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

#

# n - \u00f1 N - \u00d1

#

# Interrogación que abre - \u00bf

#

#

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

#Datos Generales del Proyecto

sonar.projectKey=mx.org.infonavit.portalempresarial

sonar.projectName=Portal Empresarial

sonar.projectVersion=1.0

sonar.scm.disabled=True

#Módulos

sonar.modules=AdminContactos,Alta\_de\_cuenta,Articulo29Patrones,AsociacionDisociacionNRP,ComprobanteFiscal,ConsultaDictamenesAutorizados,ConsultaNotificadoresEjecutores,ConsultaTrabajdores,ConsultIncidences,CumplamosJuntos,DevolucionesEmpresarial,EMPAtencionCentralizada,EMPGestorDeArchivos,EMPLibrerias,EMPMPCCModelo,EMPSeguimientoFichaPago,LoginVersionMigrada,MediosDePago,MediosDePagoWSClient,MiPerfil,OficinasAtencion,PatronesBase,PEITheme,RetrievePassword

AdminContactos.sonar.projectName=AdminContactos

Alta\_de\_cuenta.sonar.projectName=Alta\_de\_cuenta

Articulo29Patrones.sonar.projectName=Articulo29Patrones

AsociacionDisociacionNRP.sonar.projectName=AsociacionDisociacionNRP

ComprobanteFiscal.sonar.projectName=ComprobanteFiscal

ConsultaDictamenesAutorizados.sonar.projectName=ConsultaDictamenesAutorizados

ConsultaNotificadoresEjecutores.sonar.projectName=ConsultaNotificadoresEjecutores

ConsultaTrabajdores.sonar.projectName=ConsultaTrabajdores

ConsultIncidences.sonar.projectName=ConsultIncidences

CumplamosJuntos.sonar.projectName=CumplamosJuntos

DevolucionesEmpresarial.sonar.projectName=DevolucionesEmpresarial

EMPAtencionCentralizada.sonar.projectName=EMPAtencionCentralizada

EMPGestorDeArchivos.sonar.projectName=EMPGestorDeArchivos

EMPLibrerias.sonar.projectName=EMPLibrerias

EMPMPCCModelo.sonar.projectName=EMPMPCCModelo

EMPSeguimientoFichaPago.sonar.projectName=EMPSeguimientoFichaPago

LoginVersionMigrada.sonar.projectName=LoginVersionMigrada

MediosDePago.sonar.projectName=MediosDePago

MediosDePagoWSClient.sonar.projectName=MediosDePagoWSClient

MiPerfil.sonar.projectName=MiPerfil

OficinasAtencion.sonar.projectName=OficinasAtencion

PatronesBase.sonar.projectName=PatronesBase

PEITheme.sonar.projectName=PEITheme

RetrievePassword.sonar.projectName=RetrievePassword

#Sources

AdminContactos.sonar.sources=.

Alta\_de\_cuenta.sonar.sources=.

Articulo29Patrones.sonar.sources=.

AsociacionDisociacionNRP.sonar.sources=.

ComprobanteFiscal.sonar.sources=.

ConsultaDictamenesAutorizados.sonar.sources=.

ConsultaNotificadoresEjecutores.sonar.sources=.

ConsultaTrabajdores.sonar.sources=.

ConsultIncidences.sonar.sources=.

CumplamosJuntos.sonar.sources=.

DevolucionesEmpresarial.sonar.sources=.

EMPAtencionCentralizada.sonar.sources=.

EMPGestorDeArchivos.sonar.sources=.

EMPLibrerias.sonar.sources=.

EMPMPCCModelo.sonar.sources=.

EMPSeguimientoFichaPago.sonar.sources=.

LoginVersionMigrada.sonar.sources=.

MediosDePago.sonar.sources=.

MediosDePagoWSClient.sonar.sources=.

MiPerfil.sonar.sources=.

OficinasAtencion.sonar.sources=.

PatronesBase.sonar.sources=.

PEITheme.sonar.sources=.

RetrievePassword.sonar.sources=.

#Bin

AdminContactos.sonar.java.binaries=Codigo/AdminContactosCodigo\_war\_src/WEB-INF/classes

AdminContactos.sonar.java.binaries=Codigo/WorkspaceAdminContactos/WebContent/WEB-INF/classes

Alta\_de\_cuenta.sonar.java.binaries=Codigo/AccountRegistrationResponsivoCodigo.war.src/WEB-INF/classes

Alta\_de\_cuenta.sonar.java.binaries=Codigo/WorkspaceAccountRegistration/AccountRegistrationResponsivo/WebContent

Articulo29Patrones.sonar.java.binaries=Codigo/Articulo29PatronesCodigo.war.src/WEB-INF/classes

Articulo29Patrones.sonar.java.binaries=Codigo/WorkspaceArticulo29Patrones/Articulo29Patrones/WebContent/WEB-INF/classes

AsociacionDisociacionNRP.sonar.java.binaries=Codigo/AsociacionDisociacionNRPSource.war.src/WEB-INF/classes

AsociacionDisociacionNRP.sonar.java.binaries=Codigo/AsociacionDisociacionNRPWorkspace/AsociacionDisociacionNRP/WebContent/WEB-INF/classes

ComprobanteFiscal.sonar.java.binaries=Codigo/ComprobanteFiscalWorkspace/ComprobantesFiscales/target/ComprobantesFiscales-1/WEB-INF/classes

ComprobanteFiscal.sonar.java.binaries=Codigo/ComprobantesFiscalesCodigo.war.src/WEB-INF/classes

ConsultaDictamenesAutorizados.sonar.java.binaries=Codigo/ConsultaDictamenesAutorizadosSources.war.src/WEB-INF/classes

ConsultaDictamenesAutorizados.sonar.java.binaries=Codigo/ConsultaDictamenesAutorizadosWorkspace/ConsultaDictamenesAutorizados/WebContent/WEB-INF/classes

ConsultaNotificadoresEjecutores.sonar.java.binaries=Codigo/ConsultaNotificadoresEjecutoresSource.war.src/WEB-INF/classes

ConsultaNotificadoresEjecutores.sonar.java.binaries=Codigo/ConsultaNotificadoresEjecutoresWorkspace/ConsultaNotificadoresEjecutores/WebContent/WEB-INF/classes

ConsultaTrabajdores.sonar.java.binaries=Codigo/ConsultaTrabajdoresSource.war.src/WEB-INF/classes

ConsultaTrabajdores.sonar.java.binaries=Codigo/ConsultaTrabajdoresWorkSpace/ConsultaTrabajdores/WebContent/WEB-INF/classes

ConsultIncidences.sonar.java.binaries=Codigo/ConsultIncidencesCodigo.war.src/WEB-INF/classes

ConsultIncidences.sonar.java.binaries=Codigo/WorkspaceConsultIncidences/ConsultIncidences/WebContent/WEB-INF/classes

CumplamosJuntos.sonar.java.binaries=Codigo/CumplamosJuntosSource\_war\_src/WEB-INF/classes

CumplamosJuntos.sonar.java.binaries=Codigo/CumplamosJuntosWorkspace/CumplamosJuntos/WebContent/WEB-INF/classes

DevolucionesEmpresarial.sonar.java.binaries=Codigo/DevolucionesEmpresarialSource.war.src/WEB-INF/classes

DevolucionesEmpresarial.sonar.java.binaries=Codigo/DevolucionesEmpresarialWorkspace/DevolucionesEmpresarial/WebContent/WEB-INF/classes

EMPAtencionCentralizada.sonar.java.binaries=Codigo/EMPAtencionCentralizadaSource.war.src/WEB-INF/classes

EMPAtencionCentralizada.sonar.java.binaries=Codigo/EMPAtencionCentralizadaWorkspace/EMPAtencionCentralizada/WebContent/WEB-INF/classes

EMPGestorDeArchivos.sonar.java.binaries=Codigo/EMPGestorDeArchivos/bin

EMPLibrerias.sonar.java.binaries=Codigo/EMPLibrerias

EMPMPCCModelo.sonar.java.binaries=Codigo/EMPMPCCModelo/bin

EMPSeguimientoFichaPago.sonar.java.binaries=Codigo/EMPSeguimientoFichaPago/bin

LoginVersionMigrada.sonar.java.binaries=Codigo/LoginCodigo.war.src/WEB-INF/classes

LoginVersionMigrada.sonar.java.binaries=Codigo/LoginWorkspace/Login/WebContent/WEB-INF/classes

MediosDePago.sonar.java.binaries=Codigo/MediosDePagoSource.war.src/WEB-INF/classes

MediosDePago.sonar.java.binaries=Codigo/MediosDePagoWorkspace/MediosDePago/WebContent/WEB-INF/classes

MediosDePagoWSClient.sonar.java.binaries=Codigo/MediosDePagoWSClient/bin

MiPerfil.sonar.java.binaries=Codigo/MiPerfilSource.war.src/WEB-INF/classes

MiPerfil.sonar.java.binaries=Codigo/MiPerfilWorkspace/MiPerfil/WebContent/WEB-INF/classes

OficinasAtencion.sonar.java.binaries=Codigo/OficinasAtencionCodigo.war.src/WEB-INF/classes

OficinasAtencion.sonar.java.binaries=Codigo/OficinasAtencionWorkspace/OficinasAtencion/WebContent/WEB-INF/classes

PatronesBase.sonar.java.binaries=Codigo

PEITheme.sonar.java.binaries=Codigo/PEITheme\_EARFuente.ear.src/PEITheme.war.src/WEB-INF/classes

PEITheme.sonar.java.binaries=Codigo/WorkSpace\_PEITHEME/PEITheme/WebContent/WEB-INF

RetrievePassword.sonar.java.binaries=Codigo/RetrievePasswordCodigo.war.src/WEB-INF/classes

RetrievePassword.sonar.java.binaries=Codigo/Workspace\_RetrievePassword/RetrievePassword/WebContent/WEB-INF/classes

#Librerías

AdminContactos.sonar.java.libraries=Codigo/AdminContactosCodigo\_war\_src/WEB-INF/lib

AdminContactos.sonar.java.libraries=Codigo/WorkspaceAdminContactos/WebContent/WEB-INF/lib

Alta\_de\_cuenta.sonar.java.libraries=Codigo/WorkspaceAccountRegistration/AccountRegistrationResponsivo/WebContent/lib

Alta\_de\_cuenta.sonar.java.libraries=Codigo/WorkspaceAccountRegistration/AccountRegistrationResponsivo/WebContent/WEB-INF/lib

Articulo29Patrones.sonar.java.libraries=Codigo/Articulo29PatronesCodigo.war.src/WEB-INF/lib

Articulo29Patrones.sonar.java.libraries=Codigo/WorkspaceArticulo29Patrones/Articulo29Patrones/WebContent/WEB-INF/lib

AsociacionDisociacionNRP.sonar.java.libraries=Codigo/AsociacionDisociacionNRPSource.war.src/WEB-INF/lib

AsociacionDisociacionNRP.sonar.java.libraries=Codigo/AsociacionDisociacionNRPWorkspace/AsociacionDisociacionNRP/WebContent/WEB-INF/lib

ComprobanteFiscal.sonar.java.libraries=Codigo/ComprobanteFiscalWorkspace/ComprobantesFiscales/target/ComprobantesFiscales-1/WEB-INF/lib

ComprobanteFiscal.sonar.java.libraries=Codigo/ComprobantesFiscalesCodigo.war.src/WEB-INF/lib

ConsultaDictamenesAutorizados.sonar.java.libraries=Codigo/ConsultaDictamenesAutorizadosSources.war.src/WEB-INF/lib

ConsultaDictamenesAutorizados.sonar.java.libraries=Codigo/ConsultaDictamenesAutorizadosWorkspace/ConsultaDictamenesAutorizados/WebContent/WEB-INF/lib

ConsultaNotificadoresEjecutores.sonar.java.libraries=Codigo/ConsultaNotificadoresEjecutoresSource.war.src/WEB-INF/lib

ConsultaNotificadoresEjecutores.sonar.java.libraries=Codigo/ConsultaNotificadoresEjecutoresWorkspace/ConsultaNotificadoresEjecutores/WebContent/WEB-INF/lib

ConsultaTrabajdores.sonar.java.libraries=Codigo/ConsultaTrabajdoresSource.war.src/WEB-INF/lib

ConsultaTrabajdores.sonar.java.libraries=Codigo/ConsultaTrabajdoresWorkSpace/ConsultaTrabajdores/WebContent/WEB-INF/lib

ConsultIncidences.sonar.java.libraries=Codigo/ConsultIncidencesCodigo.war.src/WEB-INF/lib

ConsultIncidences.sonar.java.libraries=Codigo/WorkspaceConsultIncidences/ConsultIncidences/WebContent/WEB-INF/lib

CumplamosJuntos.sonar.java.libraries=Codigo/CumplamosJuntosSource\_war\_src/WEB-INF/lib

CumplamosJuntos.sonar.java.libraries=Codigo/CumplamosJuntosWorkspace/CumplamosJuntos/WebContent/WEB-INF/lib

DevolucionesEmpresarial.sonar.java.libraries=Codigo/DevolucionesEmpresarialSource.war.src/WEB-INF/lib

DevolucionesEmpresarial.sonar.java.libraries=Codigo/DevolucionesEmpresarialWorkspace/DevolucionesEmpresarial/WebContent/WEB-INF/lib

EMPAtencionCentralizada.sonar.java.libraries=Codigo/EMPAtencionCentralizadaSource.war.src/WEB-INF/lib

EMPAtencionCentralizada.sonar.java.libraries=Codigo/EMPAtencionCentralizadaWorkspace/EMPAtencionCentralizada/WebContent/WEB-INF/lib

EMPLibrerias.sonar.java.libraries=Codigo/EMPLibrerias/CommonsFileUpload

EMPLibrerias.sonar.java.libraries=Codigo/EMPLibrerias/PortletMinimo

LoginVersionMigrada.sonar.java.libraries=Codigo/LoginCodigo.war.src/WEB-INF/lib

LoginVersionMigrada.sonar.java.libraries=Codigo/LoginWorkspace/Login/WebContent/WEB-INF/lib

MediosDePago.sonar.java.libraries=Codigo/MediosDePagoSource.war.src/WEB-INF/lib

MediosDePago.sonar.java.libraries=Codigo/MediosDePagoWorkspace/MediosDePago/WebContent/WEB-INF/lib

MiPerfil.sonar.java.libraries=Codigo/MiPerfilSource.war.src/WEB-INF/lib

MiPerfil.sonar.java.libraries=Codigo/MiPerfilWorkspace/MiPerfil/WebContent/WEB-INF/lib

OficinasAtencion.sonar.java.libraries=Codigo/OficinasAtencionCodigo.war.src/WEB-INF/lib

OficinasAtencion.sonar.java.libraries=Codigo/OficinasAtencionWorkspace/OficinasAtencion/WebContent/WEB-INF/lib

PatronesBase.sonar.java.libraries=Codigo

PatronesBase.sonar.java.libraries=Codigo/Workspace\_PatronesLib/PatronesBase/WebContent/WEB-INF/lib

PEITheme.sonar.java.libraries=Codigo/PEITheme\_EARFuente.ear.src/PEITheme.war.src/WEB-INF/lib

PEITheme.sonar.java.libraries=Codigo/WorkSpace\_PEITHEME/PEITheme/WebContent/WEB-INF/lib

RetrievePassword.sonar.java.libraries=Codigo/RetrievePasswordCodigo.war.src/WEB-INF/lib

RetrievePassword.sonar.java.libraries=Codigo/Workspace\_RetrievePassword/RetrievePassword/WebContent/WEB-INF/lib

#Exclusions

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

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



* + 1. Ejecución de Análisis de Código Fuente

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



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



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

* + 1. Resultado del Análisis de Código Fuente

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

**Proyecto**: AdminContactos

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **"InterruptedException" no debe ser ignorada**  Int*erruptedExceptions* nunca deben ser ignoradas en el código, y simplemente registrar la excepción contando en este caso como "*ignoring*". El lanzamiento de Int*erruptedException* borra el estado interrumpido del Hilo, por lo que si la excepción no se maneja correctamente se perderá el hecho de que el hilo fue interrumpido.  En su lugar, las Int*erruptedExceptions* deben ser introducidas nuevamente o después de limpiar el estado del método o el hilo debe ser interrumpido nuevamente y llamando a *Thread.*int*errupt()* aunque se suponga que es una aplicación de un solo hilo. Cualquier otro curso de acción puede retrasar el cierre del hilo y perder la información de que el hilo fue interrumpido probablemente sin terminar su tarea.  Del mismo modo, también debe propagarse la excepción de *ThreadDeath*. Según su *JavaDoc*:  Si *ThreadDeath* es atrapado por un método, es importante que sea devuelto para que el hilo muera. | 6 |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 4 |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| *Bug* | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 2 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 8 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 8 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 6 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 28 |
| Optimizador | *Blocker* | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 8 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 4 |
| Optimizador | *Critical* | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 4 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 2 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados | 106 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 24 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 16 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 14 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 10 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 8 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 6 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 6 |
| Optimizador | *Major* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 4 |
| Optimizador | *Major* | **Los campos estáticos no deben actualizarse en constructores**  Asignar un valor a un campo estático en un constructor podría causar un comportamiento poco fiable en tiempo de ejecución ya que cambiará el valor para todas las instancias de la clase.  En su lugar, eliminar el campo modificador static o iniciar de forma estática. | 4 |
| Optimizador | *Major* | **Las sentencias "*if*" colapsables deberían fusionarse**  Fusionar sentencias *if* colapsables incrementan la legibilidad del código. | 2 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 2 |
| *Optimizador* | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 2 |
| *Optimizador* | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 86 |
| *Optimizador* | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 38 |
| *Optimizador* | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 20 |
| *Optimizador* | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 10 |
| *Optimizador* | *Minor* | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 6 |
| *Optimizador* | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 6 |
| *Optimizador* | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 4 |
| *Optimizador* | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 2 |
| *Optimizador* | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| *Optimizador* | *Minor* | **Las constantes públicas y los campos inicializados en la declaración deben ser "final estático (*static* *final*)" en lugar de meramente "*final*"**  Además, cuando un campo final no público no es también estático, implica que diferentes instancias pueden tener diferentes valores. Sin embargo, la inicialización de un campo final no estático en su declaración obliga a cada instancia a tener el mismo valor. Por lo tanto, estos campos deben ser estáticos o inicializados en el constructor. | 2 |
| *Optimizador* | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 12 |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

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

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

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

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

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

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

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

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

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

**Proyecto**: Alta\_de\_Cuenta

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 6 |
| Bug | *Major* | **"InterruptedException" no debe ser ignorada**  Int*erruptedExceptions* nunca deben ser ignoradas en el código, y simplemente registrar la excepción contando en este caso como "*ignoring*". El lanzamiento de Int*erruptedException* borra el estado interrumpido del Hilo, por lo que si la excepción no se maneja correctamente se perderá el hecho de que el hilo fue interrumpido.  En su lugar, las Int*erruptedExceptions* deben ser introducidas nuevamente o después de limpiar el estado del método o el hilo debe ser interrumpido nuevamente y llamando a *Thread.*int*errupt()* aunque se suponga que es una aplicación de un solo hilo. Cualquier otro curso de acción puede retrasar el cierre del hilo y perder la información de que el hilo fue interrumpido probablemente sin terminar su tarea.  Del mismo modo, también debe propagarse la excepción de *ThreadDeath*. Según su *JavaDoc*:  Si *ThreadDeath* es atrapado por un método, es importante que sea devuelto para que el hilo muera | 4 |
| Bug | *Major* | **No deben utilizarse expresiones idénticas en ambos lados de un operador binario**  No deben utilizarse expresiones idénticas en ambos lados de un operador binario ya que puede provocar un error.  Esta regla ignora \*, +, y = | 2 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 2 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 4 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador. | 2 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 2 |
| Vulnerabilidad | *Minor* | **Los campos mutables no deben ser "*public static*"**  No existen razones para tener un objeto mutable como “público (por defecto) *static*” en una interfaz. Estas variables deberían trasladarse a clases y su visibilidad debería reducirse.  Del mismo modo, los miembros estáticos mutables de clases y enumeraciones a los que se accede directamente, en lugar de a través de *getters* y *setters*, deben protegerse en la medida de lo posible. Esto puede hacerse reduciendo la visibilidad o haciendo que el campo sea *final*, si procede.  Tenga en cuenta que al crear un campo mutable, como un arreglo, al final evitará que la variable sea reasignada, pero al hacerlo no tiene ningún efecto sobre la mutabilidad del estado interno del arreglo (es decir, no logra la meta).  Esta regla genera problemas para los miembros de arreglos, *Collection, Date, y awt.Point* que sean *public static*. | 2 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 36 |
| Optimizador | *Blocker* | **Los retornos de métodos no deben ser invariantes**  Cuando un método está diseñado para devolver un valor invariable, puede ser un diseño deficiente, pero no debería afectar negativamente al resultado del programa. Sin embargo, cuando sucede en diferentes secciones del código, es probable que sea un error. | 2 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 16 |
| Optimizador | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 16 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 8 |
| Optimizador | *Critical* | **Las constantes no deben definirse en interfaces**  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 8 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 146 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 68 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 42 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 30 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 24 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 12 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado | 10 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 10 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 6 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 4 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 2 |
| Optimizador | *Major* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.*   *StringBuilder* en lugar de *StringBuffer.* | 2 |
| Optimizador | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 2 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 2 |
| Optimizador | *Major* | **Las asignaciones no deben ser redundantes**  La propiedad transitiva indica que si A == B y B == C, entonces A == C. En tales casos, no es necesario asignar A a C o viceversa porque ya son equivalentes. Esta regla plantea un problema cuando una asignación es inútil porque la variable asignada ya contiene el valor en todos los trayectos de ejecución. | 2 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 56 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 40 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico | 38 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 22 |
| Optimizador | *Minor* | Los indicadores de arreglos "[ ]" deben ubicarse después del tipo en los *method signatures*  En versiones anteriores de la plataforma *Java SE*, la declaración de un método que devuelve un *array* permitía colocar pares de corchetes vacíos “[ ]”para formarla declaración del tipo de *array* después de la lista de parámetros formales. Esta sintaxis no debe utilizar ya que es obsoleta. | 14 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 12 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 12 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 8 |
| Optimizador | *Minor* | Las declaraciones deben utilizar interfaces de *Java* *Collection* como "*List*" en lugar de clases de implementación específicas como "*LinkedList*"  El propósito de *Java* *Collections* *API* es proporcionar una jerarquía bien definida de interfaces para ocultar los detalles de implementación.  Las clases de implementación deben ser utilizadas para instanciar nuevas colecciones, pero el resultado de una instanciación debe ser almacenado idealmente en una variable cuyo tipo sea una interfaz de colección *Java*.  Esta regla plantea un problema cuando una clase de implementación:   * Es devuelto de un método público. * Se acepta como argumento para un método público. * Se expone como miembro público. | 6 |
| Optimizador | *Minor* | No deben declararse múltiples variables en la misma línea  Declarar múltiples variables en una línea dificulta la lectura del código. | 6 |
| Optimizador | *Minor* | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 2 |
| Optimizador | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 2 |
| Optimizador | *Minor* | **Los nombres de la interfaz deben cumplir una convención para definición de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que todos los nombres de interfaz coinciden con una expresión regular proporcionada | 2 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Los *URIs* no deben tener códigos duros (*hardcoded*)**  La codificación dura de un *URI* hace difícil probar un programa: los literales de trayectoria no siempre son portátiles a través de los sistemas operativos, una trayectoria absoluta dada puede no existir en un entorno de prueba específico, una *URL* de Internet especificada puede no estar disponible al ejecutar las pruebas, los sistemas de archivos del entorno de producción generalmente difieren del entorno de desarrollo,... etc. Por todas estas razones, un *URI* nunca debe ser codificado duro. En su lugar, debe ser reemplazado por un parámetro personalizable.  Aún más, incluso si los elementos de un *URI* se obtienen dinámicamente, la portabilidad puede ser limitada si los delimitadores de trayectoria son codificados duramente. | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

Proyecto: Artuculo29Patrones

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 8 |
| *Bug* | *Major* | **"*hashCode*" y "*toString*" no deben ser llamados en instancias de *array***  Aunque *hashCode* y *toString* están disponibles en *arreys*, son en gran parte poco útiles. *hashCode* devuelve el "código de identidad *hashCode*"del *array*, y *toString* devuelve casi el mismo valor. La salida de ninguno de los dos métodos refleja realmente el contenido del *array*. En su lugar, se debe trasladar el *array* al método pertinente de *Arrays* estáticos. | 2 |
| Optimizador | *Blocker* | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| *Optimizador* | *Critical* | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 8 |
| *Optimizador* | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 4 |
| *Optimizador* | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 108 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 94 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 14 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 12 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 12 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 8 |
| Optimizador | *Major* | **Deben eliminarse los métodos "privados" no utilizados**  Métodos privados que nunca se ejecutan son código muerto, código innecesario e inoperante que debe ser eliminado. La eliminación del código muerto disminuye el tamaño de la base de códigos mantenida, lo que facilita la comprensión del programa y evita la introducción de errores. | 8 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo | 6 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 6 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 4 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 4 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 2 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 2 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 22 |
| Optimizador | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 8 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 8 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 8 |
| Optimizador | *Minor* | **Los *URIs* no deben tener códigos duros (*hardcoded*)**  La codificación dura de un *URI* hace difícil probar un programa: los literales de trayectoria no siempre son portátiles a través de los sistemas operativos, una trayectoria absoluta dada puede no existir en un entorno de prueba específico, una *URL* de Internet especificada puede no estar disponible al ejecutar las pruebas, los sistemas de archivos del entorno de producción generalmente difieren del entorno de desarrollo,... etc. Por todas estas razones, un *URI* nunca debe ser codificado duro. En su lugar, debe ser reemplazado por un parámetro personalizable.  Aún más, incluso si los elementos de un *URI* se obtienen dinámicamente, la portabilidad puede ser limitada si los delimitadores de trayectoria son codificados duramente. | 6 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 6 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 4 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 4 |
| Optimizador | *Minor* | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones | 4 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 4 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp* | 2 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 2 |
| Optimizador | *Minor* | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

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

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

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

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

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

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

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

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

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

Proyecto: AsociacionDisociacionNRPb

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Critical* | **Las llamadas de función no deben pasar argumentos adicionales**  Es posible llamar a una función *JavaScript* con más argumentos de los necesita, pero los argumentos adicionales serán ignorados por la ejecución de la función. | 10 |
| Bug | *Major* | **No se debe acceder a las propiedades de las variables con valores "*null*" o "*undefined*"**  Cuando a una variable se le asigna un valor no *undefined* o *null*, no se generan propiedades. Intentar acceder a las propiedades de una variable de este tipo resulta en un *TypeError*, causando una terminación abrupta del *script* si el error no se detecta en un bloque de captura. | 18 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 18 |
| Bug | *Major* | **No se deben hacer llamadas a valores que no se pueden llamar**  Sintácticamente es aceptable invocar cualquier expresión como si el valor fuera una función, pero se corre el riesgo de recibir un error *TypeError*. | 8 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 8 |
| Bug | *Major* | **"*for*" las condiciones de parada del bucle deben ser invariables**  Para la condición de parada del bucle debe probar el contador del bucle contra un valor invariante (es decir, uno que es verdadero tanto al principio como al final de cada iteración del bucle). Esto significa que la condición de parada se ajusta a una variable local justo antes de que comience el bucle.  Las condiciones de parada que no son invariables son ligeramente menos eficientes, además de ser difíciles de entender y mantener, y probablemente conduzcan a la introducción de errores en el futuro. Esta regla rastrea tres tipos de condiciones de parada no invariantes:   * Cuando los contadores de bucle se actualizan en el cuerpo del bucle *for.* * Cuando la condición de parada depende de una llamada de método.   Cuando la condición de parada depende de la propiedad de un objeto, ya que estas propiedades pueden cambiar durante la ejecución del bucle. | 4 |
| Bug | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 4 |
| Bug | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 4 |
| Bug | *Major* | **Los operandos matemáticos deben unirse antes de la asignación**  Cuando una operación aritmética se realiza en números enteros, el resultado siempre será un número entero. Puede asignar ese resultado a un valor *long*, *double* o *float* con una conversión automática de tipo de dato, pero en caso de haber comenzado como *int* o *long*, es probable que el resultado no sea el esperado.  Por ejemplo, si el resultado de la división *int* se asigna a una variable de punto flotante, la precisión se habrá perdido antes de la asignación.  Para evitar resultados imprevistos por lo menos un operando debe ser lanzado o promovido al tipo | 2 |
| Bug | *Major* | **No deben utilizarse expresiones idénticas en ambos lados de un operador binario**  No deben utilizarse expresiones idénticas en ambos lados de un operador binario ya que puede provocar un error.  Esta regla ignora \*, +, y = | 2 |
| Bug | *Minor* | **Los operandos matemáticos deben unirse antes de la asignación**  Cuando una operación aritmética se realiza en números enteros, el resultado siempre será un número entero. Puede asignar ese resultado a un valor *long*, *double* o *float* con una conversión automática de tipo de dato, pero en caso de haber comenzado como *int* o *long*, es probable que el resultado no sea el esperado.  Por ejemplo, si el resultado de la división *int* se asigna a una variable de punto flotante, la precisión se habrá perdido antes de la asignación.  Para evitar resultados imprevistos por lo menos un operando debe ser lanzado o promovido al tipo de dato final antes de que la operación tenga lugar. | 6 |
| Vulnerabilidad | *Critical* | **No se deben utilizar constructores de funciones**  Los constructores de funciones aumentan el riesgo de fallo ya que su ejecución evalúa los argumentos de cadena del constructor de forma similar a la forma en que funciona la evaluación, lo que podría exponer la aplicación programa a un código aleatorio no intencionado que puede ser lento y un riesgo para la seguridad.  En general, es mejor evitarlo por completo, especialmente cuando se utiliza para analizar datos *JSON*. Se sugiere utilizar las funciones *JSON* incorporadas de *ECMAScript* 5 o una biblioteca dedicada. | 10 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 18 |
| *Optimizador* | *Blocker* | **Los retornos de función no deben ser invariantes**  Cuando una función está diseñada para devolver un valor invariable, puede ser un diseño deficiente, pero no debería afectar negativamente al resultado del programa. Sin embargo, cuando sucede en diferentes secciones del código, es probable que sea un error. | 4 |
| *Optimizador* | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 2 |
| *Optimizador* | *Blocker* | **Los *Switch* *cases* deben terminar con una declaración incondicional "*break*"**  Cuando la ejecución no se termina explícitamente al final de un *switch* *case*, continúa ejecutando las sentencias del siguiente caso. Mientras que esto es a veces intencional, a menudo es un error que lleva a un comportamiento inesperado del aplicativo. | 2 |
| *Optimizador* | *Blocker* | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimzador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 43 |
| Optimzador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 39 |
| Optimzador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 12 |
| Optimzador | *Critical* | **Las constantes no deben definirse en interfaces**  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 8 |
| Optimzador | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 4 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 252 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 180 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 112 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 91 |
| Optimizador | *Major* | **Las funciones no deben definirse dentro de los bucles**  Definir una función dentro de un bucle puede producir resultados inesperados. Esta función mantiene referencias a las variables que se definen en los alcances externos. Todas las instancias de función creadas dentro del bucle, por lo tanto, ven los mismos valores para estas variables, lo que probablemente no se espera. | 30 |
| Optimizador | *Major* | **Dos *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener dos *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 28 |
| Optimizador | *Major* | **Las sentencias de salto no deben ir seguidas de otras sentencias**  Las sentencias *Jump* (*return*, *break* y *continue*) y expresiones *throw* mueven el flujo de control fuera del bloque de código actual. | 20 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 17 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 14 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 14 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 12 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 8 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 6 |
| Optimizador | *Major* | **Las cadenas de formato Printf-style no deben provocar un comportamiento inesperado en el runtime**  Debido a que las cadenas de formato *Pr*int*f*-*style* se interpretan en el runtime en lugar de ser validadas por el compilador *Java*, pueden contener errores que provocan un comportamiento inesperado o errores durante el *runtime*. Esta regla valida de forma estática el comportamiento correcto de los formatos *Pr*int*f*-*style* cuando se llama a los métodos de formato (...) de *Java*.*util*.*Formatter*, *Java.lang.String*, *Java*.*io.Pr*int*Stream*, *MessageFormat*, y clases *Java*.*io*.*Pr*int*Writer* y de métodos *printf*(… ) de *Java*.*io.Pr*int*Stream* o clases *Java.io.Pr*int*Writer* | 6 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 4 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 121 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 112 |
| Optimizador | *Minor* | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 90 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 62 |
| Optimizador | *Minor* | Las declaraciones deben utilizar interfaces de *Java* *Collection* como "*List*" en lugar de clases de implementación específicas como "*LinkedList*"  El propósito de *Java* *Collections* *API* es proporcionar una jerarquía bien definida de interfaces para ocultar los detalles de implementación.  Las clases de implementación deben ser utilizadas para instanciar nuevas colecciones, pero el resultado de una instanciación debe ser almacenado idealmente en una variable cuyo tipo sea una interfaz de colección *Java*.  Esta regla plantea un problema cuando una clase de implementación:   * Es devuelto de un método público. * Se acepta como argumento para un método público. * Se expone como miembro público. | 58 |
| Optimizador | *Minor* | **Las declaraciones vacías deben ser eliminadas**  Las afirmaciones vacías, es decir las que se introducen generalmente por error deben eliminarse. | 50 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 30 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 28 |
| Optimizador | *Minor* | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 26 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 20 |
| Optimizador | *Minor* | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 20 |
| Optimizador | *Minor* | **Los bloques de multilínea deben estar encerrados en llaves**  Las llaves pueden omitirse de un bloque de una sola línea, por ejemplo con una sentencia *if* o para bucle, pero hacerlo puede ser engañoso e inducir a errores.  Esta regla plantea un problema cuando el espaciado en blanco de las líneas después de un bloque de una línea indica la intención de incluir esas líneas en el bloque, pero la omisión de las llaves significa que las líneas se ejecutarán incondicionalmente una vez. | 16 |
| Optimizador | *Minor* | **La devolución de expresiones booleanas no debe ser envuelta en una declaración *"if-then-else"***  Debe simplificarse la devolución de expresiones literales booleanas envueltas en enunciados *if-then-else*. | 14 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 12 |
| Optimizador | *Minor* | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 12 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

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

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

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

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

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

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

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

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

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

Proyecto: ComprobanteFiscal

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 1 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 5 |
| *Optimizador* | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 1 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 52 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 40 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 16 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 6 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 2 |
| Optimizador | *Minor* | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones. | 2 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 2 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 1 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

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

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

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

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

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

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

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

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

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

Proyecto: ConsultaDictamenesAutorizados

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 6 |
| *Bug* | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 2 |
| Optimizador | *Blocker* | **Los métodos y los nombres de campo no deben ser iguales o diferir sólo por la capitalización**  Visualizar el conjunto de métodos en una clase, incluyendo los métodos de superclase, y encontrar dos métodos o campos que sólo difieren por capitalización es confuso para los usuarios de la clase. Es igualmente confuso tener un método y un campo que difieren sólo en mayúsculas o un método y un campo con exactamente el mismo nombre y visibilidad.  En el caso de los métodos, puede haber sido un error por parte del desarrollador original, quien intentó anular un método de superclase, pero en su lugar agregó un nuevo método con casi el mismo nombre. De lo contrario, esta situación indica una mala denominación. Los nombres de los métodos deben estar orientados a la acción y, por lo tanto, contener un verbo, lo cual es improbable en el caso de que tanto un método como un miembro tengan el mismo nombre (con o sin diferencias de capitalización). Sin embargo, cambiar el nombre de un método público podría ser perjudicial para las entidades que mandan llamar al método. Por lo tanto, cambiar el nombre del miembro es la acción recomendada. | 2 |
| Optimizador | *Blocker* | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | *Blocker* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 10 |
| Optimizador | *Blocker* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 8 |
| Optimizador | *Blocker* | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 4 |
| Optimizador | *Blocker* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada. | 2 |
| Optimizador | *Blocker* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada. | 2 |
| Optimizador | *Blocker* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 60 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 28 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 8 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 6 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 4 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo | 4 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 4 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 4 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | *Major* | **Las declaraciones "*switch*" deben tener por lo menos 3 cláusulas "*case*"**  Las declaraciones “*Switch*” son útiles cuando hay muchos casos diferentes dependiendo del valor de la misma expresión. Sin embargo, sólo en uno o dos casos, el código será más legible con sentencias *if*. | 2 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

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

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

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

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

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

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

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

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

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

Proyecto: ConsultaNotificadoresEjecutadores

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| *Bug* | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 2 |
| Vulnerabilidad | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 2 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 2 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 14 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 14 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 6 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 4 |
| Optimizador | *Critical* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 114 |
| Optimizador | *Critical* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 46 |
| Optimizador | *Critical* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 12 |
| Optimizador | *Critical* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 6 |
| Optimizador | *Critical* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 5 |
| Optimizador | *Critical* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 3 |
| Optimizador | *Critical* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 2 |
| Optimizador | *Critical* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 2 |
| Optimizador | *Critical* | **Los nombres de las propiedades no deben ser duplicados dentro de una clase u objeto**  *JavaScript* permite duplicar el nombre de las propiedades en las clases y objetos pero sólo la última instancia de un nombre duplicado determina el valor real que será utilizado. Por lo tanto, cambiar los valores de otras ocurrencias de un nombre duplicado no tendrá ningún efecto y puede causar errores. | 2 |
| Optimizador | *Critical* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Optimizador | *Critical* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 2 |
| Optimizador | *Critical* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 2 |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

Proyecto: ConsultaTrabjdores

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 6 |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 8 |
| *Bug* | *Major* | **"InterruptedException" no debe ser ignorada**  Int*erruptedExceptions* nunca deben ser ignoradas en el código, y simplemente registrar la excepción contando en este caso como "*ignoring*". El lanzamiento de Int*erruptedException* borra el estado interrumpido del Hilo, por lo que si la excepción no se maneja correctamente se perderá el hecho de que el hilo fue interrumpido.  En su lugar, las Int*erruptedExceptions* deben ser introducidas nuevamente o después de limpiar el estado del método o el hilo debe ser interrumpido nuevamente y llamando a *Thread.*int*errupt()* aunque se suponga que es una aplicación de un solo hilo. Cualquier otro curso de acción puede retrasar el cierre del hilo y perder la información de que el hilo fue interrumpido probablemente sin terminar su tarea.  Del mismo modo, también debe propagarse la excepción de *ThreadDeath*. Según su *JavaDoc*:  Si *ThreadDeath* es atrapado por un método, es importante que sea devuelto para que el hilo muera. | 2 |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador. | 2 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 2 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 2 |
| Vulnerabilidad | *Minor* | Las excepciones no deben ser arrojadas desde los métodos de *servlets*  Aunque las firmas de los métodos en un *servlet* incluyen *IOException*, *ServletException*, no se recomienda dejar que tales excepciones sean lanzadas.  El mal manejo de tales excepciones podría dejar al sistema en un estado vulnerable, resultando en ataques de denegación de servicio o la exposición de información sensible, esto debido a que cuando un *servlet* lanza una excepción, el contenedor envía la información de depuración al usuario, y ésta podría abre una brecha para un atacante.  Esta regla verifica que todas las excepciones en los métodos "*do\*"* se manejan explícitamente en las clases *servlet*. | 2 |
| Optimizador | *Blocker* | **Los nombres de las clases no deberían ensombrecer interfaces o superclases**  Aunque es posible asignar a una clase el mismo nombre de una clase ubicada en otro paquete que amplía la interfaz que implementa, es confuso y podría causar problemas en el futuro. | 2 |
| Optimizador | *Blocker* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 30 |
| Optimizador | *Blocker* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 14 |
| Optimizador | *Blocker* | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 4 |
| Optimizador | *Blocker* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 4 |
| Optimizador | *Blocker* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 156 |
| Optimizador | *Blocker* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 14 |
| Optimizador | *Blocker* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 12 |
| Optimizador | *Blocker* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 10 |
| Optimizador | *Blocker* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 8 |
| Optimizador | *Blocker* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 6 |
| Optimizador | *Blocker* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 4 |
| Optimizador | *Blocker* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 4 |
| Optimizador | *Blocker* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 4 |
| Optimizador | *Blocker* | **Los métodos de anulación deben hacer más que simplemente llamar al mismo método en la clase *super* (*super* *class*)**  Anular un método sólo para llamar al mismo método desde la superclase sin realizar ninguna otra acción es inútil y engañoso. La única vez que esto se justifica es en los métodos de anulación final, donde el efecto es bloquear el comportamiento de la clase padre. Esta regla ignora anulaciones tales como *equals*, *hashCode* y *toString.* | 4 |
| Optimizador | *Blocker* | **Los métodos no deberían tener implementaciones idénticas**  La duplicidad en la implementación de métodos generalmente se presenta por un error o la intención explicita de duplicar la implementación. La duplicidad de implementación de métodos puede ser confusa para el equipo responsable de dar mantenimiento al código por lo que se recomienda eliminar la duplicidad e implementar la invocación de una aplicación a otra. | 4 |
| Optimizador | *Blocker* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | *Blocker* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 2 |
| Optimizador | *Blocker* | **Un campo no debe duplicar el nombre de la clase que contiene**  Es confuso tener a un miembro de la clase con el mismo nombre (diferencias de caso aparte) que su clase cerrada. Esto es particularmente así cuando se considera la práctica común de nombrar una instancia de clase para la clase misma. La mejor práctica dicta que cualquier campo o miembro con el mismo nombre que la clase cerrada sea renombrada para ser más descriptivo del aspecto particular de la clase que representa o tiene | 2 |
| Optimizador | *Blocker* | **Los campos estáticos no deben actualizarse en constructores**  Asignar un valor a un campo estático en un constructor podría causar un comportamiento poco fiable en tiempo de ejecución ya que cambiará el valor para todas las instancias de la clase.  En su lugar, eliminar el campo modificador static o iniciar de forma estática. | 2 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 114 |
| Optimizador | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 60 |
| Optimizador | *Minor* | No deben declararse múltiples variables en la misma línea  Declarar múltiples variables en una línea dificulta la lectura del código. | 20 |
| Optimizador | *Minor* | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones. | 6 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 4 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 4 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 4 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | ***Collection.isEmpty()* debe usarse para probar vacíos**  *Collection.size ()* se usa para probar los trabajos vacíos, pero el uso de *Collection.isEmpty* () hace que el código sea más legible y más eficaz. La complejidad temporal de cualquier implementación del método *isEmpty* () debe ser *O(1)* considerando que algunas implementaciones de tamaño () pueden ser *O(n*). | 2 |
| Optimizador | *Minor* | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 2 |
| Optimizador | *Minor* | **Los métodos de anulación deben hacer más que simplemente llamar al mismo método en la clase *super* (*super* *class*)**  Anular un método sólo para llamar al mismo método desde la superclase sin realizar ninguna otra acción es inútil y engañoso. La única vez que esto se justifica es en los métodos de anulación final, donde el efecto es bloquear el comportamiento de la clase padre. Esta regla ignora anulaciones tales como *equals*, *hashCode* y *toString.* | 2 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 2 |
| Optimizador | *Minor* | **Las declaraciones "*switch*" deben tener por lo menos 3 cláusulas "*case*"**  Las declaraciones “*Switch*” son útiles cuando hay muchos casos diferentes dependiendo del valor de la misma expresión. Sin embargo, sólo en uno o dos casos, el código será más legible con sentencias *if*. | 2 |
| Optimizador | *Minor* | **No se deben hacer verificaciones de igualdad**  Las comparaciones de diferentes tipos siempre dan como resultado *false*. Por lo que las comparaciones y el código dependiente debe eliminarse. Esto incluye:   * Comparando un objeto con *null.* * Comparando un objeto con un primitivo no emparentado (E. G. *string* con un int). * Comparación de clases no relacionadas. * Comparación de una clase e interfaz no relacionada. * Comparación de tipos de interfaz no relacionados. * Comparando un *array* con un *no-array*. * Comparando dos *arrays.*   Específicamente en el caso de los *arrays*, dado que los *arrays* no anulan *Objects.equals(),* llamar iguales en dos *arrays* es lo mismo que comparar sus direcciones. Esto significa que *array1*.*equals*(*array2)* es equivalente a *array1*===*array2*.  Sin embargo, algunos equipos de desarrollo pueden esperar que *Array.equals(Object obj*) haga más que una simple comparación de direcciones de memoria, comparando por ejemplo el tamaño y contenido de los dos *arrays*. En su lugar, == *operator* o *Arrays.equals(array1, array2*) siempre debe ser usado con *arrays*. | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

Proyecto: ConsultIncidences

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 14 |
| Bug | *Critical* | **Las llamadas de función no deben pasar argumentos adicionales**  Es posible llamar a una función *JavaScript* con más argumentos de los necesita, pero los argumentos adicionales serán ignorados por la ejecución de la función. | 10 |
| Bug | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 87 |
| Bug | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 46 |
| Bug | *Major* | **No se debe acceder a las propiedades de las variables con valores "*null*" o "*undefined*"**  Cuando a una variable se le asigna un valor no *undefined* o *null*, no se generan propiedades. Intentar acceder a las propiedades de una variable de este tipo resulta en un *TypeError*, causando una terminación abrupta del *script* si el error no se detecta en un bloque de captura. | 18 |
| Bug | *Major* | **"InterruptedException" no debe ser ignorada**  Int*erruptedExceptions* nunca deben ser ignoradas en el código, y simplemente registrar la excepción contando en este caso como "*ignoring*". El lanzamiento de Int*erruptedException* borra el estado interrumpido del Hilo, por lo que si la excepción no se maneja correctamente se perderá el hecho de que el hilo fue interrumpido.  En su lugar, las Int*erruptedExceptions* deben ser introducidas nuevamente o después de limpiar el estado del método o el hilo debe ser interrumpido nuevamente y llamando a *Thread.*int*errupt()* aunque se suponga que es una aplicación de un solo hilo. Cualquier otro curso de acción puede retrasar el cierre del hilo y perder la información de que el hilo fue interrumpido probablemente sin terminar su tarea.  Del mismo modo, también debe propagarse la excepción de *ThreadDeath*. Según su *JavaDoc*:  Si *ThreadDeath* es atrapado por un método, es importante que sea devuelto para que el hilo muera. | 14 |
| Bug | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 10 |
| Bug | *Major* | **No se deben hacer llamadas a valores que no se pueden llamar**  Sintácticamente es aceptable invocar cualquier expresión como si el valor fuera una función, pero se corre el riesgo de recibir un error *TypeError*. | 8 |
| Bug | *Major* | **"*for*" las condiciones de parada del bucle deben ser invariables**  Para la condición de parada del bucle debe probar el contador del bucle contra un valor invariante (es decir, uno que es verdadero tanto al principio como al final de cada iteración del bucle). Esto significa que la condición de parada se ajusta a una variable local justo antes de que comience el bucle.  Las condiciones de parada que no son invariables son ligeramente menos eficientes, además de ser difíciles de entender y mantener, y probablemente conduzcan a la introducción de errores en el futuro. Esta regla rastrea tres tipos de condiciones de parada no invariantes:   * Cuando los contadores de bucle se actualizan en el cuerpo del bucle *for.* * Cuando la condición de parada depende de una llamada de método.   Cuando la condición de parada depende de la propiedad de un objeto, ya que estas propiedades pueden cambiar durante la ejecución del bucle. | 4 |
| Bug | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 4 |
| Bug | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 4 |
| Bug | *Major* | **No deben utilizarse expresiones idénticas en ambos lados de un operador binario**  No deben utilizarse expresiones idénticas en ambos lados de un operador binario ya que puede provocar un error.  Esta regla ignora \*, +, y = | 4 |
| Bug | *Major* | **"*hashCode*" y "*toString*" no deben ser llamados en instancias de *array***  Aunque *hashCode* y *toString* están disponibles en *arreys*, son en gran parte poco útiles. *hashCode* devuelve el "código de identidad *hashCode*"del *array*, y *toString* devuelve casi el mismo valor. La salida de ninguno de los dos métodos refleja realmente el contenido del *array*. En su lugar, se debe trasladar el *array* al método pertinente de *Arrays* estáticos. | 4 |
| Bug | *Major* | **No se deben utilizar operadores de igualdad estricta con tipos diferentes**  Realizar comparaciones de tipos diferentes utilizando operadores de igualdad estricta como === y !== siempre devolverá el mismo valor ya que no se realiza ninguna conversión de tipo antes de la comparación, por lo que se deduce se trata de un bug inmerso en el código. | 2 |
| Bug | *Major* | **Bloques ejecutados condicionalmente deben ser accesibles**  Las expresiones condicionales que son siempre verdaderas o falsas pueden conducir a código muerto. Dicho código siempre es considerado un *bug* y nunca debe usarse en la producción. | 2 |
| Bug | *Major* | **Las excepciones no deben ser arrojadas a bloques finales**  Lanzar una excepción desde dentro de un bloque final enmascarará cualquier excepción que haya sido lanzada previamente en el *try* o *catchblock*, y se perderá el mensaje de excepción del enmascarado y el rastreo de pila. | 2 |
| Bug | *Minor* | **Los operandos matemáticos deben unirse antes de la asignación**  Cuando una operación aritmética se realiza en números enteros, el resultado siempre será un número entero. Puede asignar ese resultado a un valor *long*, *double* o *float* con una conversión automática de tipo de dato, pero en caso de haber comenzado como *int* o *long*, es probable que el resultado no sea el esperado.  Por ejemplo, si el resultado de la división *int* se asigna a una variable de punto flotante, la precisión se habrá perdido antes de la asignación.  Para evitar resultados imprevistos por lo menos un operando debe ser lanzado o promovido al tipo de dato final antes de que la operación tenga lugar. | 6 |
| Vulnerabilidad | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 23 |
| Vulnerabilidad | *Critical* | **No se deben utilizar constructores de funciones**  Los constructores de funciones aumentan el riesgo de fallo ya que su ejecución evalúa los argumentos de cadena del constructor de forma similar a la forma en que funciona la evaluación, lo que podría exponer la aplicación programa a un código aleatorio no intencionado que puede ser lento y un riesgo para la seguridad.  En general, es mejor evitarlo por completo, especialmente cuando se utiliza para analizar datos *JSON*. Se sugiere utilizar las funciones *JSON* incorporadas de *ECMAScript* 5 o una biblioteca dedicada. | 10 |
| Vulnerabilidad | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 49 |
| Vulnerabilidad | *Minor* | Las variables de una clase no deben tener accesibilidad pública  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador. | 42 |
| Vulnerabilidad | *Minor* | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 24 |
| Vulnerabilidad | *Minor* | Las excepciones no deben ser arrojadas desde los métodos de *servlets*  Aunque las firmas de los métodos en un *servlet* incluyen *IOException*, *ServletException*, no se recomienda dejar que tales excepciones sean lanzadas.  El mal manejo de tales excepciones podría dejar al sistema en un estado vulnerable, resultando en ataques de denegación de servicio o la exposición de información sensible, esto debido a que cuando un *servlet* lanza una excepción, el contenedor envía la información de depuración al usuario, y ésta podría abre una brecha para un atacante.  Esta regla verifica que todas las excepciones en los métodos "*do\*"* se manejan explícitamente en las clases *servlet*. | 8 |
| Vulnerabilidad | *Minor* | **Los campos mutables no deben ser "*public static*"**  No existen razones para tener un objeto mutable como “público (por defecto) *static*” en una interfaz. Estas variables deberían trasladarse a clases y su visibilidad debería reducirse. | 2 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 88 |
| Optimizador | *Blocker* | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 28 |
| Optimizador | *Blocker* | **Los métodos y los nombres de campo no deben ser iguales o diferir sólo por la capitalización**  Visualizar el conjunto de métodos en una clase, incluyendo los métodos de superclase, y encontrar dos métodos o campos que sólo difieren por capitalización es confuso para los usuarios de la clase. Es igualmente confuso tener un método y un campo que difieren sólo en mayúsculas o un método y un campo con exactamente el mismo nombre y visibilidad.  En el caso de los métodos, puede haber sido un error por parte del desarrollador original, quien intentó anular un método de superclase, pero en su lugar agregó un nuevo método con casi el mismo nombre. De lo contrario, esta situación indica una mala denominación. Los nombres de los métodos deben estar orientados a la acción y, por lo tanto, contener un verbo, lo cual es improbable en el caso de que tanto un método como un miembro tengan el mismo nombre (con o sin diferencias de capitalización). Sin embargo, cambiar el nombre de un método público podría ser perjudicial para las entidades que mandan llamar al método. Por lo tanto, cambiar el nombre del miembro es la acción recomendada. | 18 |
| Optimizador | *Blocker* | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 14 |
| Optimizador | *Blocker* | **Los retornos de función no deben ser invariantes**  Cuando una función está diseñada para devolver un valor invariable, puede ser un diseño deficiente, pero no debería afectar negativamente al resultado del programa. Sin embargo, cuando sucede en diferentes secciones del código, es probable que sea un error. | 4 |
| Optimizador | *Blocker* | **Los *Switch* *cases* deben terminar con una declaración incondicional "*break*"**  Cuando la ejecución no se termina explícitamente al final de un *switch* *case*, continúa ejecutando las sentencias del siguiente caso. Mientras que esto es a veces intencional, a menudo es un error que lleva a un comportamiento inesperado del aplicativo. | 2 |
| Optimizador | *Blocker* | **Los retornos de métodos no deben ser invariantes**  Cuando un método está diseñado para devolver un valor invariable, puede ser un diseño deficiente, pero no debería afectar negativamente al resultado del programa. Sin embargo, cuando sucede en diferentes secciones del código, es probable que sea un error. | 2 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 267 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 145 |
| Optimizador | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 76 |
| Optimizador | *Critical* | **Las constantes no deben definirse en interfaces**  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 51 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 49 |
| Optimizador | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase.  **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 15 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 8 |
| Optimizador | *Critical* | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 6 |
| Optimizador | *Critical* | **Los tipos de *wildcard* genéricos no *deben* utilizarse en los parámetros de return**  Se recomienda no utilizar *wildcards* como tipos de devolución. Debido a que las reglas de inferencia de tipo son bastante complejas, es poco probable que el usuario de esa *API* sepa cómo utilizarla correctamente. El uso de tipos de *wildcards* debe limitarse a los parámetros del método. | 2 |
| Optimizador | *Critical* | **Los nombres de las clases no deberían ensombrecer interfaces o superclases**  Aunque es posible asignar a una clase el mismo nombre de una clase ubicada en otro paquete que amplía la interfaz que implementa, es confuso y podría causar problemas en el futuro. | 1 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 2118 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 725 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas. | 203 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 200 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 169 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 132 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 118 |
| Optimizador | *Major* | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 117 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 103 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 102 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 85 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 74 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 70 |
| Optimizador | *Major* | **Dos *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener dos *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 64 |
| Optimizador | *Major* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 59 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 783 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 318 |
| Optimizador | *Minor* | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 280 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 274 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 260 |
| Optimizador | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 200 |
| Optimizador | *Minor* |  | 124 |
| Optimizador | *Minor* |  | 104 |
| Optimizador | *Minor* |  | 96 |
| Optimizador | *Minor* |  | 90 |
| Optimizador | *Minor* |  | 88 |
| Optimizador | *Minor* |  | 85 |
| Optimizador | *Minor* |  | 78 |
| Optimizador | *Minor* |  | 67 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

Proyecto: CumplamosJuntos

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 20 |
| Optimizador | *Blocker* | **Los nombres de las clases no deberían ensombrecer interfaces o superclases**  Aunque es posible asignar a una clase el mismo nombre de una clase ubicada en otro paquete que amplía la interfaz que implementa, es confuso y podría causar problemas en el futuro. | 4 |
| Optimizador | *Critical* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 40 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 34 |
| Optimizador | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 14 |
| Optimizador | *Critical* | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 4 |
| Optimizador | *Major* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 290 |
| Optimizador | *Major* | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 60 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 34 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 18 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 16 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 8 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 6 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 6 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 6 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 6 |
| Optimizador | *Major* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 4 |
| Optimizador | *Major* | **Las llamadas de función no deben pasar argumentos adicionales**  Es posible llamar a una función *JavaScript* con más argumentos de los necesita, pero los argumentos adicionales serán ignorados por la ejecución de la función. | 4 |
| Optimizador | *Major* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 2 |
| Optimizador | *Major* | **Los métodos no deberían tener implementaciones idénticas**  La duplicidad en la implementación de métodos generalmente se presenta por un error o la intención explicita de duplicar la implementación. La duplicidad de implementación de métodos puede ser confusa para el equipo responsable de dar mantenimiento al código por lo que se recomienda eliminar la duplicidad e implementar la invocación de una aplicación a otra. | 2 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 184 |
| Optimizador | *Minor* | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 174 |
| Optimizador | *Minor* | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 106 |
| Optimizador | *Minor* | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 54 |
| Optimizador | *Minor* | No deben declararse múltiples variables en la misma línea  Declarar múltiples variables en una línea dificulta la lectura del código. | 26 |
| Optimizador | *Minor* | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 20 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 6 |
| Optimizador | *Minor* | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 6 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 6 |
| Optimizador | *Minor* | **Las constantes públicas y los campos inicializados en la declaración deben ser "final estático (*static* *final*)" en lugar de meramente "*final*"**  Además, cuando un campo final no público no es también estático, implica que diferentes instancias pueden tener diferentes valores. Sin embargo, la inicialización de un campo final no estático en su declaración obliga a cada instancia a tener el mismo valor. Por lo tanto, estos campos deben ser estáticos o inicializados en el constructor. | 6 |
| Optimizador | *Minor* | **Los métodos "private" que no acceden a los datos de instancia deben ser "static"**  Los métodos "*private*" que no acceden a los datos de instancia pueden ser estáticos para evitar cualquier malentendido sobre el contrato del método. | 6 |
| Optimizador | *Minor* | **Las comparaciones de cadenas insensibles a mayúsculas y minúsculas deben realizarse sin carcasa intermedia superior o inferior**  El uso de *toLowerCase*() o *toUpperCase* () para hacer comparaciones insensibles al caso es ineficiente porque requiere la creación de objetos *String* temporales e intermedios. | 2 |
| Optimizador | *Minor* | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 2 |
| Optimizador | *Minor* | **Los métodos de anulación deben hacer más que simplemente llamar al mismo método en la clase *super* (*super* *class*)**  Anular un método sólo para llamar al mismo método desde la superclase sin realizar ninguna otra acción es inútil y engañoso. La única vez que esto se justifica es en los métodos de anulación final, donde el efecto es bloquear el comportamiento de la clase padre. Esta regla ignora anulaciones tales como *equals*, *hashCode* y *toString.* | 2 |
| Optimizador | *Minor* | **Los indicadores de arreglos "[ ]" deben estar en el tipo, no en la variable**  Los indicadores de arreglos deben estar siempre localizados en el tipo para una mejor legibilidad del código. De lo contrario, los desarrolladores deben mirar tanto el tipo como el nombre de la variable para saber si una variable es o no un *array*. | 2 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

Proyecto: [DevolucionesEmpresarial](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3ADevolucionesEmpresarial)

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 24 |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 10 |
| *Bug* | *Major* | **Dos *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener dos *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto | 4 |
| *Bug* | *Major* | **"*hashCode*" y "*toString*" no deben ser llamados en instancias de *array***  Aunque *hashCode* y *toString* están disponibles en *arreys*, son en gran parte poco útiles. *hashCode* devuelve el "código de identidad *hashCode*"del *array*, y *toString* devuelve casi el mismo valor. La salida de ninguno de los dos métodos refleja realmente el contenido del *array*. En su lugar, se debe trasladar el *array* al método pertinente de *Arrays* estáticos. | 2 |
| Optimizador | *Blocker* | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 16 |
| Optimizador | *Blocker* | **Los métodos y los nombres de campo no deben ser iguales o diferir sólo por la capitalización**  Visualizar el conjunto de métodos en una clase, incluyendo los métodos de superclase, y encontrar dos métodos o campos que sólo difieren por capitalización es confuso para los usuarios de la clase. Es igualmente confuso tener un método y un campo que difieren sólo en mayúsculas o un método y un campo con exactamente el mismo nombre y visibilidad.  En el caso de los métodos, puede haber sido un error por parte del desarrollador original, quien intentó anular un método de superclase, pero en su lugar agregó un nuevo método con casi el mismo nombre. De lo contrario, esta situación indica una mala denominación. Los nombres de los métodos deben estar orientados a la acción y, por lo tanto, contener un verbo, lo cual es improbable en el caso de que tanto un método como un miembro tengan el mismo nombre (con o sin diferencias de capitalización). Sin embargo, cambiar el nombre de un método público podría ser perjudicial para las entidades que mandan llamar al método. Por lo tanto, cambiar el nombre del miembro es la acción recomendada. | 12 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 18 |
| Optimizador | *Critical* | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 6 |
| Optimizador | *Critical* | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 4 |
| Optimizador | *Critical* | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 4 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 282 |
| Optimizador | *Major* | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 230 |
| Optimizador | *Major* | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 34 |
| Optimizador | *Major* | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado | 28 |
| Optimizador | *Major* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 16 |
| Optimizador | *Major* | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 14 |
| Optimizador | *Major* | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 12 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 10 |
| Optimizador | *Major* | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 8 |
| Optimizador | *Major* | ***Deben eliminarse los métodos "privados" no utilizados***  Métodos privados que nunca se ejecutan son código muerto, código innecesario e inoperante que debe ser eliminado. La eliminación del código muerto disminuye el tamaño de la base de códigos mantenida, lo que facilita la comprensión del programa y evita la introducción de errores | 8 |
| Optimizador | *Major* | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión | 6 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 4 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 4 |
| Optimizador | *Major* | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 2 |
| Optimizador | *Major* | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 2 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 60 |
| Optimizador | *Minor* | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 50 |
| Optimizador | *Minor* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 10 |
| Optimizador | *Minor* | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones. | 8 |
| Optimizador | *Minor* | **Los *URIs* no deben tener códigos duros (*hardcoded*)**  La codificación dura de un *URI* hace difícil probar un programa: los literales de trayectoria no siempre son portátiles a través de los sistemas operativos, una trayectoria absoluta dada puede no existir en un entorno de prueba específico, una *URL* de Internet especificada puede no estar disponible al ejecutar las pruebas, los sistemas de archivos del entorno de producción generalmente difieren del entorno de desarrollo,... etc. Por todas estas razones, un *URI* nunca debe ser codificado duro. En su lugar, debe ser reemplazado por un parámetro personalizable.  Aún más, incluso si los elementos de un *URI* se obtienen dinámicamente, la portabilidad puede ser limitada si los delimitadores de trayectoria son codificados duramente. | 6 |
| Optimizador | *Minor* | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 6 |
| Optimizador | *Minor* | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 6 |
| Optimizador | *Minor* | ***Collection.isEmpty()* debe usarse para probar vacíos**  *Collection.size ()* se usa para probar los trabajos vacíos, pero el uso de *Collection.isEmpty* () hace que el código sea más legible y más eficaz. La complejidad temporal de cualquier implementación del método *isEmpty* () debe ser *O(1)* considerando que algunas implementaciones de tamaño () pueden ser *O(n*). | 4 |
| Optimizador | *Minor* | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 4 |
| Optimizador | *Minor* | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 2 |
| Optimizador | *Minor* | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | *Minor* | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 2 |
| Optimizador | *Minor* | **Los nombres de las clases no deberían ensombrecer interfaces o superclases**  Aunque es posible asignar a una clase el mismo nombre de una clase ubicada en otro paquete que amplía la interfaz que implementa, es confuso y podría causar problemas en el futuro. | 2 |
| Optimizador | *Minor* | **Métodos de "*Random*" que devuelven valores de punto flotante no deben ser usados en la generación de enteros aleatorios**  No hay necesidad de multiplicar la salida del método *nextDouble* de *Random* para obtener un entero aleatorio. Utilice en su lugar el método *nextInt*. Esta regla plantea un problema cuando el valor de retorno de cualquier método de *Random* que devuelve un valor de coma flotante se convierte a un entero. | 2 |
| Optimizador | *Minor* | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

Proyecto: [EMPAtencionCentralizada](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3AEMPAtencionCentralizada)

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 2 |
| Bug | *Blocker* | **Las Variables deben ser declaradas de forma explicita**  La creación de variables globales se da cuando se declara una variable dentro de una función o la cláusula “*for*” en de un bucle *for-loop* utilizando palabras clave como *let*, *const* o *var.* | 6 |
| Bug | Critical | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 2 |
| Bug | Critical | **Las constantes no deben definirse en interfaces**  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la API exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 2 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 20 |
| Optimizador | *Major* | **Dos *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener dos *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto | 18 |
| Optimizador | *Major* | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 10 |
| Optimizador | *Major* | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 4 |
| Optimizador | *Major* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 2 |
| Optimizador | *Major* | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 2 |
| Optimizador | *Major* | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 2 |
| Optimizador | *Minor* | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 16 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto:** [**EMPGestorDeArchivos**](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3AEMPGestorDeArchivos)

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Optimizador* | *Critical* | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 1 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 2 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico | 4 |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto:** [**EMP**](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3AEMPGestorDeArchivos)Librerías

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
|  |  |  |  |
| Bug |  |  | 0 |
| Vulnerabilidad |  |  | 0 |
| Optimizador |  |  | 0 |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto:** [**EMP**](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3AEMPGestorDeArchivos)MPCCModelo

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 6 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto:** [EMPSeguimientoFichaPago](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3AEMPSeguimientoFichaPago)

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Vulnerabilidad* | *Blocker* | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 2 |
| Optimizador | Critical | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 2 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 7 |
| Optimizador | Major | **"Preconditions" y los argumentos de registro no deberían requerir evaluación**  Pasar los argumentos de mensaje que requieren una evaluación adicional en una *Guava.com.google.common.base.Preconditionscheck* puede resultar en una penalización de rendimiento. Esto se debe a que, independientemente de si es necesario o no, cada argumento debe resolverse antes de que se llame realmente al método.  Del mismo modo, pasar cadenas concatenadas a un método de registro también puede tener un impacto innecesario en el rendimiento debido a que las concatenaciones se realizarán cada vez que se llame al método, independientemente de que el nivel de registro sea lo suficientemente bajo como para mostrar el mensaje. En su lugar, se debe estructurar el código para pasar valores estáticos o pre-calculados en condiciones de verificación de condiciones previas y registro de llamadas.  Específicamente, el formato de cadena incorporado debe ser usado en lugar de concatenación de cadena, y si el mensaje es el resultado de una llamada de método, entonces las precondiciones deben ser omitidas en alto, y la excepción relevante debe ser lanzada condicionalmente en su lugar. | 6 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 14 |
| Optimizador | Minor | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 4 |
| Optimizador | Minor | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones | 2 |
| Optimizador | Minor | **Métodos de "*Random*" que devuelven valores de punto flotante no deben ser usados en la generación de enteros aleatorios**  No hay necesidad de multiplicar la salida del método *nextDouble* de *Random* para obtener un entero aleatorio. Utilice en su lugar el método *nextInt*. Esta regla plantea un problema cuando el valor de retorno de cualquier método de *Random* que devuelve un valor de coma flotante se convierte a un entero. | 1 |
| Optimizador | Minor | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 1 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto: LoginVersionMigrada**

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Vulnerabilidad* | *Blocker* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 12 |
| Vulnerabilidad | Blocker | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 12 |
| Optimizador | Blocker | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 6 |
| Optimizador | Blocker | **Los métodos y los nombres de campo no deben ser iguales o diferir sólo por la capitalización**  Visualizar el conjunto de métodos en una clase, incluyendo los métodos de superclase, y encontrar dos métodos o campos que sólo difieren por capitalización es confuso para los usuarios de la clase. Es igualmente confuso tener un método y un campo que difieren sólo en mayúsculas o un método y un campo con exactamente el mismo nombre y visibilidad.  En el caso de los métodos, puede haber sido un error por parte del desarrollador original, quien intentó anular un método de superclase, pero en su lugar agregó un nuevo método con casi el mismo nombre. De lo contrario, esta situación indica una mala denominación. Los nombres de los métodos deben estar orientados a la acción y, por lo tanto, contener un verbo, lo cual es improbable en el caso de que tanto un método como un miembro tengan el mismo nombre (con o sin diferencias de capitalización). Sin embargo, cambiar el nombre de un método público podría ser perjudicial para las entidades que mandan llamar al método. Por lo tanto, cambiar el nombre del miembro es la acción recomendada. | 4 |
| Optimizador | Blocker | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | Critical | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3 | 32 |
| Optimizador | Critical | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase | 6 |
| Optimizador | Critical | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 4 |
| Optimizador | Critical | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada. | 2 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 80 |
| Optimizador | Major | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 34 |
| Optimizador | Major | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 28 |
| Optimizador | Major | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 22 |
| Optimizador | Major | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 16 |
| Optimizador | Major | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 10 |
| Optimizador | Major | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 4 |
| Optimizador | Major | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 4 |
| Optimizador | Major | **Los bloques *Try-catch* no deben estar anidados**  Anidar *try/catch* bloques afecta gravemente a la legibilidad del código fuente porque hace demasiado difícil entender qué bloque atrapará cada excepción. | 4 |
| Optimizador | Major | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 4 |
| Optimizador | Major | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 2 |
| Optimizador | Major | **“*@Override*" se debe utilizar en los métodos de ejecución y de anulación**  El uso de la anotación @*Override* es útil por dos razones:   * Se obtiene una advertencia del compilador si el método anotado no anula nada, como en el caso de un error ortográfico. * Mejora la legibilidad del código fuente al hacer evidente que los métodos son anulados. | 2 |
| Optimizador | Major | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 34 |
| Optimizador | Major | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 22 |
| Optimizador | Major | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 20 |
| Optimizador | Major | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 18 |
| Optimizador | Major | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 6 |
| Optimizador | Major | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 6 |
| Optimizador | Major | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 4 |
| Optimizador | Major | **Los métodos de anulación deben hacer más que simplemente llamar al mismo método en la clase *super* (*super* *class*)**  Anular un método sólo para llamar al mismo método desde la superclase sin realizar ninguna otra acción es inútil y engañoso. La única vez que esto se justifica es en los métodos de anulación final, donde el efecto es bloquear el comportamiento de la clase padre. Esta regla ignora anulaciones tales como *equals*, *hashCode* y *toString.* | 4 |
| Optimizador | Major | **Los indicadores de arreglos "[ ]" deben ubicarse después del tipo en los method signatures**  En versiones anteriores de la plataforma *Java SE*, la declaración de un método que devuelve un *array* permitía colocar pares de corchetes vacíos “[ ]”para formarla declaración del tipo de *array* después de la lista de parámetros formales. Esta sintaxis no debe utilizar ya que es obsoleta. | 4 |
| Optimizador | Major | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 4 |
| Optimizador | Major | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 4 |
| Optimizador | Major | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 4 |
| Optimizador | Major | **Las declaraciones vacías deben ser eliminadas**  Las afirmaciones vacías, es decir las que se introducen generalmente por error deben eliminarse | 2 |
| Optimizador | Major | **Los *URIs* no deben tener códigos duros (*hardcoded*)**  La codificación dura de un *URI* hace difícil probar un programa: los literales de trayectoria no siempre son portátiles a través de los sistemas operativos, una trayectoria absoluta dada puede no existir en un entorno de prueba específico, una *URL* de Internet especificada puede no estar disponible al ejecutar las pruebas, los sistemas de archivos del entorno de producción generalmente difieren del entorno de desarrollo,... etc. Por todas estas razones, un *URI* nunca debe ser codificado duro. En su lugar, debe ser reemplazado por un parámetro personalizable.  Aún más, incluso si los elementos de un *URI* se obtienen dinámicamente, la portabilidad puede ser limitada si los delimitadores de trayectoria son codificados duramente. | 2 |
| Optimizador | Major | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto: MediosDePago**

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 4 |
| Bug | Major | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 2 |
| Optimizador | Blocker | **Las variables deberán declararse explícitamente**  Debe evitarse la creación de variables globales. Este escenario ocurre cuando se declara una variable dentro de una función o la cláusula *for* de un bucle *for-loop* sin keywords como *let, const* o *var*. | 6 |
| Optimizador | Blocker | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | Critical | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 6 |
| Optimizador | Critical | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 2 |
| Optimizador | Critical | **Los tipos de *wildcard* genéricos no *deben* utilizarse en los parámetros de return**  Se recomienda no utilizar *wildcards* como tipos de devolución. Debido a que las reglas de inferencia de tipo son bastante complejas, es poco probable que el usuario de esa *API* sepa cómo utilizarla correctamente. El uso de tipos de *wildcards* debe limitarse a los parámetros del método. | 2 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 188 |
| Optimizador | Major | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 40 |
| Optimizador | Major | **Dos *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener dos *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 18 |
| Optimizador | Major | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 14 |
| Optimizador | Major | **Deberían eliminarse los *dead store***  Una *dead* *store* ocurre cuando a una variable local se le asigna un valor, incluyendo nulo, que no es leído por ninguna instrucción posterior. Calcular o recuperar un valor sólo para luego sobre escribirlo o desecharlo, podría indicar un error grave en el código. Aunque no sea un error, representa un desperdicio de recursos.  Incluso la asignación nula a una variable es un almacén muerto si la variable no se utiliza posteriormente. Asignar nulo como una pista al recolector de basura solía ser una práctica común, pero ya no es necesario y ese código debería ser eliminado. | 10 |
| Optimizador | Major | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 2 |
| Optimizador | Major | **Las variables y funciones no deben volver a declararse**  Esta regla verifica que una declaración no utilice un nombre que ya está en uso. Es posible utilizar el mismo símbolo varias veces como variable o como función, pero hacerlo puede confundir al equipo responsable de dar mantenimiento al código. Adicional, es posible que tales reasignaciones se hagan por error, sin que el equipo de desarrollo se dé cuenta de que el valor de la variable es sobrescrito por la nueva asignación.  Esta regla también se aplica a los parámetros de función. | 2 |
| Optimizador | Major | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 2 |
| Optimizador | Major | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 2 |
| Optimizador | Major | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 2 |
| Optimizador | Major | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 2 |
| Optimizador | Minor | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 44 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 34 |
| Optimizador | Minor | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 30 |
| Optimizador | Minor | **Las variables locales y funciones locales no utilizadas deben ser eliminadas**  Si se declara una variable local o una función local pero no se utiliza, es considerado código muerto y debe eliminarse. Hacerlo mejorará la mantenibilidad del código. | 16 |
| Optimizador | Minor | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 6 |
| Optimizador | Minor | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código. | 6 |
| Optimizador | Minor | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 6 |
| Optimizador | Minor | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 4 |
| Optimizador | Minor | **El código deprecado debe ser eliminado**  Esta regla está pensada para ser usada como una forma de rastrear código que está marcado como obsoleto. El código deprecado debe ser eventualmente eliminado. | 4 |
| Optimizador | Minor | **Las declaraciones vacías deben ser eliminadas**  Las afirmaciones vacías, es decir las que se introducen generalmente por error deben eliminarse | 2 |
| Optimizador | Minor | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 2 |
| Optimizador | Minor | **Las declaraciones "*switch*" deben tener por lo menos 3 cláusulas "*case*"**  Las declaraciones “*Switch*” son útiles cuando hay muchos casos diferentes dependiendo del valor de la misma expresión. Sin embargo, sólo en uno o dos casos, el código será más legible con sentencias *if*. | 2 |
| Optimizador | Minor | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones. | 2 |
| Optimizador | Minor | **Las clases no serializables no deben ser escritas**  Intentar serializar una clase no serializable tiene como resultado una excepción. Únicamente una clase que implementa la interfaz *Serializable* y sus extensiones pueden ser serializables de forma satisfactoria. | 2 |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto: MediosDePagoWSClient**

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Optimizador* | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 12 |
| *Optimizador* | *Critical* | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 11 |
| *Optimizador* | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 22 |
| Optimizador | Minor | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 60 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 58 |
|  |  |  |  |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto: MiPerfil**

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action* | 2 |
| *Bug* | *Major* | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Vulnerabilidad | Blocker | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 4 |
| Optimizador | Blocker | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | Critical | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 8 |
| Optimizador | Critical | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 6 |
| Optimizador | Critical | Las constantes no deben definirse en interfaces  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 2 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 44 |
| Optimizador | Major | **Las clases internas anónimas que contienen sólo un método deben convertirse en Lambdas**  Antes de la versión 8 de *Java*, la única manera de soportar parcialmente los cierres en *Java* era utilizando clases internas anónimas. Pero la sintaxis de las clases anónimas puede parecer difícil de manejar y poco clara. | 6 |
| Optimizador | Major | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario | 2 |
| Optimizador | Major | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 2 |
| Optimizador | Major | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 2 |
| Optimizador | Major | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 2 |
| Optimizador | Minor | **Los nombres de los paquetes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de los paquetes coincidan con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z\_]+(\.[a-z\_][a-z0-9\_]\*)\*$* | 6 |
| Optimizador | Minor | **Los *Null Pointers* deben estar referenciados**  Una referencia a un *null* nunca debe ser desreferenciada/accedida. En caso de realizarse, se lanzará una *NullPo*int*erException*. En el mejor de los casos, tal excepción causará la terminación abrupta del programa. En el peor de los casos, podría exponer información de depuración que sería útil para un atacante, o podría permitir a un atacante eludir las medidas de seguridad.  Se debe considerar que cuando están presentes, esta regla aprovecha las anotaciones @*CheckForNull* y @*Nonnull* definidas en *JSR*-*305* para entender qué valores son y cuáles no son nulos excepto cuando se utiliza @*Nonnull* en el parámetro a igual, que por contrato siempre debe trabajar con *null*. | 6 |
| Optimizador | Minor | **Los nombres de las constantes deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres constantes coincidan con una expresión regular proporcionada.  Utilizar la siguiente expresión regular por defecto: ^[A-Z][A-Z0-9]\*(\_[A-Z0-9]+)\*$: | 6 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 6 |
| Optimizador | Minor | **Las expresiones booleanas deben cambiar la evaluación de la condición**  Si una expresión booleana no cambia la evaluación de la condición, entonces es completamente innecesaria y puede ser eliminada. | 4 |
| Optimizador | Minor | **Las variables locales no utilizadas deben eliminarse**  Si una variable local es declarada pero no utilizada, es código muerto y debe ser eliminada. Esto mejorará la mantenibilidad porque los desarrolladores no se preguntarán para qué sirve la variable. | 2 |
| Optimizador | Minor | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá | 2 |
| Optimizador | Minor | **Las capturas deberían combinarse**  Desde la versión 7 de *Java* ha sido posible capturar múltiples excepciones a la vez. Por lo tanto, cuando varios bloques de captura tienen el mismo código, deben combinarse para una mejor legibilidad. | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto: OficinasAtencion**

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 2 |
| Optimizador | Blocker | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | Critical | **Los campos de una clase "*Serializable*" deben ser transitorios o serializables**  Datos no transitorios y no serializables podrían causar bloqueos del programa y abrir la puerta a los atacantes. En general, se espera que una clase *Serializable* cumpla con su contrato y no tenga un comportamiento inesperado cuando una instancia es serializada.    Esta regla plantea un problema en los campos no *Serializable* y en los campos de colección cuando no son privados (porque se les pueden asignar valores no serializables externamente) y cuando se les asignan tipos no serializables dentro de la clase. | 8 |
| Optimizador | Critical | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 6 |
| Optimizador | Critical | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 4 |
| Optimizador | Critical | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 4 |
| Optimizador | Critical | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 54 |
| Optimizador | Major | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 8 |
| Optimizador | Major | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado | 6 |
| Optimizador | Major | **Los métodos no deben tener demasiados parámetros**  Una larga lista de parámetros puede indicar que se debe crear una nueva estructura para completar los numerosos parámetros o que la función está haciendo demasiadas cosas. Se deben utilizar como máximo de 4 parámetros. | 4 |
| Optimizador | Major | **Los *arrays* vacíos y las colecciones deben ser devueltos en lugar de nulos**  Devolver *null* en lugar de un *array* real o coleccionar fuerza a los que llaman del método a probar explícitamente la nulidad, haciéndolos más complejos y menos legibles. Adicional, en muchos casos, *null* se utiliza como sinónimo de vacío. | 2 |
| Optimizador | Major | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 2 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 8 |
| Optimizador | Minor | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 6 |
| Optimizador | Minor | **Las constantes públicas y los campos inicializados en la declaración deben ser "final estático (*static* *final*)" en lugar de meramente "*final*"**  Además, cuando un campo final no público no es también estático, implica que diferentes instancias pueden tener diferentes valores. Sin embargo, la inicialización de un campo final no estático en su declaración obliga a cada instancia a tener el mismo valor. Por lo tanto, estos campos deben ser estáticos o inicializados en el constructor. | 6 |
| Optimizador | Minor | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 6 |
| Optimizador | Minor | **Las declaraciones vacías deben ser eliminadas**  Las afirmaciones vacías, es decir las que se introducen generalmente por error deben eliminarse | 2 |
| Optimizador | Minor | **Rastrear el uso de *tags* "TODO”**  Los *tags* “TODO” generalmente son utilizados para marcar ubicaciones en dónde se colocará más código en etapas posteriores del desarrollo.  Esta regla se genera con el objetivo de rastrear los *tags* y asegurar que no se mantengan en el código | 8 |
|  |  |  |  |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto: PatronesBase**

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Blocker* | **Los recursos deben estar cerrados**  Las conexiones, secuencias, archivos y otras clases que implementan la interfaz *Closeable* o *AutoCloseable*, deben cerrarse después de su uso. Además, esa llamada cerrada debe hacerse en un bloque final, de lo contrario una excepción podría impedir que se haga la llamada. Preferiblemente, cuando la clase implementa “*AutoCloseable*”, el recurso debe ser creado usando el patrón "*try*-*with*-*resources*" y se cerrará automáticamente. Si no se cierran correctamente los recursos, se producirá una fuga de recursos. | 1 |
| Bug | Major | **Los punteros nulos (*Null Pointers*) no deben ser descodificados**  Una referencia a un *null* nunca debe ser descodificada/accedida. En caso de accesar, se lanzará una excepción del tipo *NullPointerException*. Tal excepción causará la terminación abrupta del programa o en el escenario más drástico, podría exponer información de depuración que sería útil para un atacante o podría permitir a un atacante eludir las medidas de seguridad. | 2 |
| Vulnerabilidad | Blocker | **Las credenciales (*Password*) no deben estar en código duro**  Debido a la facilidad de extraer cadenas de una aplicación compilada, las credenciales nunca deben estar en código duro. De ser así, está casi garantizado que terminarán en manos de un atacante. Esto es especialmente cierto en el caso de las aplicaciones distribuidas.  Las credenciales deben almacenarse fuera del código en un archivo de configuración o base de datos cifrado y fuertemente protegido. | 3 |
| Vulnerabilidad | Minor | **Las variables de una clase no deben tener accesibilidad pública**  Las variables públicas en una clase no respetan el principio de encapsulamiento y tienen 3 desventajas principales:   * No se puede agregar Comportamiento adicional como la validación. * La representación interna está expuesta y no puede modificarse posteriormente. * Los valores de las variables están sujetos a cambios desde cualquier parte del código y pueden no cumplir con las suposiciones del programador.   Mediante el uso de atributos privados y métodos de acceso (*set* y *get*), se evitan las modificaciones no autorizadas. | 12 |
| Vulnerabilidad | Minor | **Los campos "*public static*" deben ser constantes**  No se recomienda declarar un campo "*public*" y "*static*" sin declararlo "*final*". Esto es un *kludge* para compartir un estado entre varios objetos, pero con este enfoque, cualquier objeto modificará el estado compartido, como cambiarlo a *null*. | 12 |
| Vulnerabilidad | Minor | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 8 |
| Optimizador | Critical | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 6 |
| Optimizador | Critcal | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 5 |
| Optimizador | Critcal | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada.   El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco | 4 |
| Optimizador | Critcal | **Las declaraciones “*switch*" deben terminar con cláusulas “*default*”**  El requisito para una cláusula final “*defect*” es un ejemplo de programación defensiva. La cláusula debería adoptar las medidas apropiadas o contener un comentario adecuado sobre la razón por la que no se ha adoptado alguna medida. | 2 |
| Optimizador | Critcal | ***Las constantes no deben definirse en interfaces***  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 1 |
| Optimizador | Critcal | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 1 |
| Optimizador | Major | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 31 |
| Optimizador | Major | **Las sentencias "*if*" colapsables deberían fusionarse**  Fusionar sentencias *if* colapsables incrementan la legibilidad del código. | 19 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 15 |
| Optimizador | Major | **El árbol de herencia de las clases no debe ser demasiado profundo**  La herencia es ciertamente uno de los conceptos más valiosos en la programación orientada a objetos. Es una manera de compartimentar y reutilizar el código creando colecciones de atributos y comportamientos llamados clases que pueden basarse en clases creadas previamente. Pero abusar de este concepto creando un árbol de herencia profundo puede llevar a un código fuente muy complejo e inalterable. La mayoría de las veces un árbol de herencia demasiado profundo se debe a un mal diseño orientado a objetos que ha llevado al uso sistemático de `herencia' cuando, por ejemplo, `composición' se adaptaría mejor. | 11 |
| Optimizador | Major | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 10 |
| Optimizador | Major | **Los bloques anidados de código no deben dejarse vacíos**  Los bloques de código vacíos tanto deben llenarse o retirarse. | 7 |
| Optimizador | Major | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 6 |
| Optimizador | Major | **El uso de la función *String* debe optimizarse para caracteres individuales**  Una llamada a *indexOf* o *lastIndexOf* con una sola letra *String* se puede hacer más eficaz cambiando a una llamada con un argumento *char.* | 4 |
| Optimizador | Major | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 3 |
| Optimizador | Major | **Bloques de código no deben ser "comentados"**  Los programadores no deben comentar el código, ya que bloquea los programas y reduce la legibilidad. El código no utilizado debe ser borrado y puede ser recuperado del historial de control de fuente si es necesario. | 2 |
| Optimizador | Major | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 2 |
| Optimizador | Major | **Los métodos no deberían tener implementaciones idénticas**  La duplicidad en la implementación de métodos generalmente se presenta por un error o la intención explicita de duplicar la implementación. La duplicidad de implementación de métodos puede ser confusa para el equipo responsable de dar mantenimiento al código por lo que se recomienda eliminar la duplicidad e implementar la invocación de una aplicación a otra. | 2 |
| Optimizador | Major | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación | 1 |
| Optimizador | Major | **Las salidas estándar no deben utilizarse directamente para generar *logs***  Al grabar en *log* un mensaje hay varias necesidades importantes que deben cumplirse:   * El usuario debe ser capaz de recuperar fácilmente los registros. * El formato de todos los mensajes registrados debe ser uniforme para permitir al usuario leer fácilmente el registro. * Los datos deben ser registrados. * Los datos confidenciales sólo deben registrarse de forma segura.   Si un programa escribe directamente en las salidas estándar, no hay forma de cumplir con estos requisitos. Es por eso que es altamente recomendable definir y usar un registrador dedicado. | 1 |
| Optimizador | Major | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión. | 1 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico. | 34 |
| Optimizador | Minor | **Los modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 22 |
| Optimizador | Minor | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 17 |
| Optimizador | Minor | **Los nombres de los campos deben cumplir con una convención de nombres**  Compartir algunas convenciones de nomenclatura es un punto clave para que un equipo pueda colaborar eficazmente. Esta regla permite comprobar que los nombres de campo coinciden con una expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$* | 14 |
| Optimizador | Minor | **Los nombres de campo estáticos no finales deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que los nombres de campo estáticos no finales coincidan con una expresión regular proporcionada. | 14 |
| Optimizador | Minor | **No deben declararse múltiples variables en la misma línea**  Declarar múltiples variables en una línea dificulta la lectura del código. | 12 |
| Optimizador | Minor | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 9 |
| Optimizador | Minor | **El código "*@Deprecated*" no debe usarse**  Una vez depreciados, las clases e interfaces, y sus miembros deben ser evitadas, en lugar de ser utilizados, heredados o extendidos. La depreciación es una advertencia de que la clase o interfaz ha sido reemplazada, y eventualmente se eliminará. El período de depreciación le permite hacer una transición suave para salir de la tecnología de envejecimiento, que pronto se retirará. | 6 |
| Optimizador | Minor | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá. | 6 |
| Optimizador | Minor | **Los nombres de los métodos deben cumplir con una convención de nomenclatura**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla verifica que todos los nombres de método coincidan con una expresión regular proporcionada.  Con la expresión regular proporcionada por defecto *^[a-z][a-zA-Z0-9]\*$:* | 4 |
| Optimizador | Minor | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 3 |
| Optimizador | Minor | **No se deben crear *arrays* para parámetros *varargs***  No es necesario crear una matriz únicamente con el propósito de pasarla como argumento *varargs* (...); *varargs* es una matriz. Es más eficaz pasar los elementos directamente y se consolidarán en un array automáticamente. | 3 |
| Optimizador | Minor | **Los nombres de clase deben cumplir con una convención de nombres**  Las convenciones de codificación compartida permiten a los equipos colaborar eficazmente. Esta regla permite comprobar que todos los nombres de clase coinciden con una expresión regular.  Con la expresión regular proporcionada por defecto *^[A-Z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | Minor | **Los literales booleanos no deberían ser redundantes**  Los literales booleanos redundantes deben eliminarse de las expresiones para mejorar la legibilidad. | 2 |
| Optimizador | Minor | **Las declaraciones "*switch*" deben tener por lo menos 3 cláusulas "*case*"**  Las declaraciones “*Switch*” son útiles cuando hay muchos casos diferentes dependiendo del valor de la misma expresión. Sin embargo, sólo en uno o dos casos, el código será más legible con sentencias *if*. | 2 |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto:** [**PEITheme**](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3APEITheme)

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Vulnerabilidad* | *Minor* | ***Throwable.printStackTrace(...)* no debe ser utilizado**  *Throwable.printStackTrace(...)* imprime un *Throwable* y su *stack trace* a algún *stream*. Por defecto, ese *streamSystem.Err* podría exponer inadvertidamente información sensible.  Se recomienda el uso de *Loggers* en lugar de imprimir *Throwables*, ya que tienen algunas ventajas:   * Los usuarios pueden recuperar fácilmente los *logs*. * El formato de los mensajes de log es uniforme y permite a los usuarios navegar fácilmente por ellos.   Esta regla genera un problema cuando *printStackTrace* se usa sin argumentos, es decir, cuando el *stack trace* se imprime en el *stream* predeterminado. | 4 |
| Optimizador | Critical | **La complejidad cognitiva de los métodos no debe ser demasiado alta**  La Complejidad Cognitiva es una medida que indica la dificultad de interpretar el flujo de control de un método. Los métodos con alta Complejidad Cognitiva son difíciles de mantener. | 2 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 16 |
| Optimizador | Major | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 8 |
| Optimizador | Major | **Declaraciones plegables de "*if*" deben fusionarse**  Se deben fusionar declaraciones “*if*” plegable si las sentencias aumentan la legibilidad del código. | 2 |
| Optimizador | Major | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.* * *StringBuilder* en lugar de *StringBuffer.* | 2 |
| Optimizador | Major | **"entrySet()" debe iterarse cuando se necesitan tanto la clave como el valor**  Cuando sólo se necesitan las teclas de un mapa en un bucle, iterar el keySet tiene sentido. Pero cuando se necesita tanto la clave como el valor, es más eficiente iterar el entrySet, ya que dará acceso tanto a la clave como al valor. | 2 |
| Optimizador | Minor | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico | 8 |
| Optimizador | Minor | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 2 |
| Optimizador | Minor | **La devolución de expresiones booleanas no debe ser envuelta en una declaración *"if-then-else"***  Debe simplificarse la devolución de expresiones literales booleanas envueltas en enunciados *if-then-else*. | 2 |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

# **Proyecto:** [**RetrievePassword**](http://10.80.1.146:9000/sonarqube/dashboard?id=mx.org.infonavit.portalempresarial%3ARetrievePassword)

Ilustración Gráfico de *Bug*s

Ilustración Gráfico de Vulnerabilidades

Ilustración Gráfico de Optimizadores

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

| **Descripción de Incidencias** | | | |
| --- | --- | --- | --- |
| **Tipo** | **Severidad** | **Incidencia** | **Cantidad de Incidencias** |
| *Bug* | *Major* | **Los *Servlets* no deberían tener campos de instancia mutable**  Por contrato, un contenedor de *servlets* crea una instancia de cada *servlet* y luego se adjunta un hilo dedicado a cada nueva petición *HTTP* entrante para procesar esta petición. Así que todos los hilos están compartiendo las instancias de *servlet* y por extensión los campos de instancia. Para evitar alguna mala interpretación y comportamiento inesperado en tiempo de ejecución, todos los campos de *servlet* deben ser estáticos y/o finales, o simplemente eliminados. Con *Struts* 1. X, la misma restricción existe en *org.apache.struts.action.Action*. | 1 |
| Optimizador | Blocker | **La lógica de cortocircuito debe usarse en contextos booleanos**  Debe utilizar la lógica de Cortocircuito en un contexto booleano ya que de lo contrarioes probable que se provoquen errores de programa a medida que las condiciones se evalúan en las circunstancias equivocadas. | 2 |
| Optimizador | Blocker | **Los campos de las clases hijo no deben ensombrecer los campos de la clase padre**  Tener una variable con el mismo nombre dentro de una jerarquía de clase genera confusión. | 2 |
| Optimizador | Critical | **Los métodos de instancia no deben escribir en campos *"static”***  Actualizar correctamente un campo estático desde un método no estático es difícil de corregir y podría fácilmente llevar a errores si hay múltiples instancias de clase y/o múltiples hilos en juego. Idealmente, los campos estáticos sólo se actualizan a partir de métodos estáticos sincronizados. | 4 |
| Optimizador | Critical | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 3 |
| Optimizador | Critical | **Las constantes no deben definirse en interfaces**  El patrón de interfaz constante es un mal uso de las interfaces. El hecho de que una clase utilice algunas constantes internamente es un detalle de implementación. La implementación de una interfaz constante hace que este detalle de implementación se filtre en la *API* exportada de la clase. Si en una futura versión la clase se modifica para que ya no necesite utilizar las constantes, debe implementar la interfaz para asegurar la compatibilidad binaria. Si una clase no final implementa una interfaz constante, todas sus subclases tendrán sus espacios de nombres contaminados por las constantes en la interfaz. | 2 |
| Optimizador | Critical | **Los métodos no deben estar vacíos**  Hay varias razones para que un método no tenga cuerpo de método:   * Es una omisión involuntaria, y debe ser corregida para evitar un comportamiento inesperado en la producción. * Todavía no se apoya, o nunca se apoyará. En este caso, se debe lanzar una excepción de operación no soportada. * El método es una anulación intencionalmente en blanco. En este caso, un comentario anidado debe explicar el motivo de la anulación en blanco. | 1 |
| Optimizador | Major | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 70 |
| Optimizador | Major | **Los bloques deben estar sincronizados en campos "*private final*"**  La sincronización en un campo de clase no se sincroniza en el campo, sino en el objeto que se le ha asignado. Por lo tanto, la sincronización en un campo no final hace posible que el valor del campo cambie mientras una hebra está en un bloque sincronizado con el valor antiguo. Esto permitiría que un segundo hilo sincronizado en el nuevo valor entre en el bloque al mismo tiempo.  Un caso similar de da cuando dos hilos diferentes ejecutando el método en paralelo podrían pasar dos instancias de objeto diferentes al método como parámetros, socavando completamente la sincronización. | 30 |
| Optimizador | Major | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 12 |
| Optimizador | Major | **Campos "*private*" que no son utilizados deben removerse**  Si un campo privado es declarado, pero no utilizado en el programa, puede ser considerado código muerto y por lo tanto debe ser eliminado.  Obsérvese que esta regla no tiene en cuenta la reflexión, lo que significa que se plantearán cuestiones sobre los campos privados a los que sólo se accede mediante la *API* de reflexión | 7 |
| Optimizador | Major | **Los constructores no deben ser utilizados para instanciar las clases "*String*" y *primitive*-*wrapper***  Los constructores de *Strings* y los objetos utilizados para envolver primitivos nunca deben utilizarse. Hacerlo es menos claro y usa más memoria que simplemente usar el valor deseado en el caso de los *strings*, y usar *valueOf* para todo lo demás. | 6 |
| Optimizador | Major | **Las clases *Utility* no deben tener constructores públicos**  Las clases *Utility*, son colecciones de miembros estáticos que no se consideran para ser instanciadas. Incluso las clases de *Utility* abstractas que pueden ampliarse, no deberían tener constructores públicos.  Java añade un constructor público implícito. Por lo tanto, debe definirse al menos un constructor no público. | 4 |
| Optimizador | Major | ***Throwable* y *Error* no deben ser capturados**  *Throwable* es la superclase de todos los errores y excepciones en *Java*. Error es la superclase de todos los errores, que no deben ser capturados por las aplicaciones. La captura de *Throwable* o Error también capturará *OutOfMemoryError* e *InternalError*, de los cuales una aplicación no debe intentar recuperarse. | 4 |
| Optimizador | Major | **Los parámetros de métodos no utilizados deben ser eliminados**  Los parámetros no utilizados son engañosos. Cualesquiera que sean los valores pasados a dichos parámetros, el comportamiento será el mismo. | 1 |
| Optimizador | Major | **Deben eliminarse las importaciones que no son utilizadas**  La parte de importaciones de un archivo debe ser manejada por el *Integrated Development Environment* (*IDE*), no manualmente por el desarrollador. Las importaciones no utilizadas no deberían producirse si ese es el caso. Dejarlos dentro reduce la legibilidad del código, ya que su presencia puede ser confusa. | 35 |
| Optimizador | Major | **Los nombres de parámetros de variables locales y métodos deben cumplir con una convención de nombres**  Las convenciones de nomenclatura compartida permiten a los equipos colaborar eficazmente. Esta regla plantea un problema cuando una variable local o un nombre de parámetro de función no coincide con la expresión regular proporcionada.  Con la expresión regular predeterminada *^[a-z][a-zA-Z0-9]\*$:* | 3 |
| Optimizador | Major | **El operador diamante ("<>") debe ser utilizado**  La versión 7 de *Java* introdujo el operador de diamantes (<>) para simplificar el código genérico | 3 |
| Optimizador | Major | L **os modificadores deben ser declarados en el orden correcto**  La especificación de lenguaje *Java* recomienda listar los modificadores en el siguiente orden:   1. *Annotations* 2. *Public* 3. *Protected* 4. *Private* 5. *Abstract* 6. *Static* 7. *Final* 8. *Transient* 9. *Volatile* 10. *Synchronized* 11. *Native* 12. *Strictfp*   No seguir esta convención no tiene ningún impacto técnico, pero reducirá la legibilidad del código porque la mayoría de los equipos de desarrollo están acostumbrados al orden estándar. | 2 |
| Optimizador | Major | **Las declaraciones "*throws*" no deberían ser superfluas**  Una excepción en una declaración de lanzamientos en *Java* es superflua si:   * Aparece con frecuencia. * Una subclase de otra excepción de la lista. * Una *RuntimeException*, o uno de sus descendientes. * Completamente innecesario porque el tipo de excepción declarado no se puede lanzar realmente. | 2 |
| Optimizador | Major | **Las clases de excepción deben ser inmutables**  Las excepciones tienen por objeto representar el estado de la aplicación en el momento en que se produjo un error. La realización de todos los campos en una clase de excepción final asegura que este estado:   * Se definirá completamente al mismo tiempo que se instancie la Excepción. * No será actualizado o corrompido por un gestor de errores cuestionable.   Esto permitirá al equipo responsable de mantenimiento al código comprender rápidamente la incidencia. | 2 |
| Optimizador | Major | **No deben declararse múltiples variables en la misma línea**  Declarar múltiples variables en una línea dificulta la lectura del código. | 2 |
| Optimizador | Major | **Los campos privados utilizados únicamente como variables locales en los métodos deben convertirse en variables locales**  Cuando el valor de un campo privado se asigna siempre en métodos de clase antes de ser leído, no se utiliza para almacenar información de clase. Por lo tanto, debería convertirse en una variable local en los métodos pertinentes para evitar confusiones. | 1 |
| Optimizador | Major | **Las variables locales no deben ser declaradas y luego devueltas o lanzadas inmediatamente**  Declarar una variable sólo para devolverla o desecharla inmediatamente es una mala práctica. Esta variable es un detalle interno de implementación que no está expuesto a los que llaman al método. El nombre del método debe ser suficiente para que las personas que llamen sepan exactamente lo que se devolverá | 1 |
| Optimizador | Major | **No deben utilizarse moldes (casts) redundantes**  Expresiones innecesarias hacen que el código sea más difícil de leer y entender. | 1 |
|  |  |  |  |
|  |  |  |  |

El detalle completo del Análisis de Código Fuente puede ser consultado en la siguiente liga: [*http://10.80.1.146:9000/sonarqube7/projects*](http://10.80.1.146:9000/sonarqube7/projects)

* 1. Alineación Entre Diseño y Construcción de Soluciones Tecnológicas

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

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

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

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

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

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

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

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

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

* 1. Cumplimiento Documental para Liberación en Ambiente Productivo

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

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

* 1. Resultado Integral

El resultado global es una vista de alto nivel sobre las observaciones que surgieron a partir de la evaluación de la solución tecnológica. Esta vista es generada por medio de los indicadores generados por cada uno de los rubros evaluados y un resultado integral considerado el porcentaje de valor de cada rubro.

* **Indicadores de Evaluación de Calidad de Código**
  + **Resultado General:** Este indicador muestra la cantidad total de líneas de código del proyecto y la cantidad de incidencias (*Bugs*, Vulnerabilidades y Optimizadores) encontradas por medio del análisis de calidad de código.

Ilustración Resultado General - Código

Ilustración Resultado General - Incidencias

* + **Comparación entre Versiones:** Este indicador muestra la comparativa entre en análisis de calidad de código aplicado a la **última versión desplegada del proyecto** (Versión operando en ambiente productivo y la **versión actualizada más reciente** (Versión con modificaciones recientes que aún no ha sido desplegada en ambiente productivo).

La evaluación de la solución tecnológica consideró únicamente la última versión desplegada en ambiente productivo ya que no se identifica una versión más reciente del proyecto.

Ilustración Gráfico Comparativo - Código

Ilustración Comparativo – Incidencias General

Ilustración Gráfico Comparativo de *Bug*s

Ilustración Gráfico Comparativo de Vulnerabilidades

Ilustración Gráfico Comparativo de Optimizadores

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

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

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

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

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

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

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

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

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

1. 9 Propuestas de Acciones de Mejora

* **Calidad de Código**

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

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

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

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

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

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

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

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

1. Conclusiones

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

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