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

**Anexo Reporte de Evaluación de Solución Tecnológica - Aclaraciones**

**15/oct/2018**

**Contenido**

[1. Objetivo 3](#_Toc519840867)

[2. Introducción 3](#_Toc519840868)

[3. Alcance 4](#_Toc519840869)

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

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

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

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

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

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

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

[4.7. Resultado Integral 13](#_Toc519840877)

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

[5. Conclusiones 19](#_Toc519840879)

1. Objetivo

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

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

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

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

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

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

1. Introducción

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

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

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



Ilustración 1 Propuesta de Valor

1. Alcance

La evaluación de soluciones tecnológicas considera los siguientes aspectos:

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

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

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

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

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

No se cuenta con información sobre el aplicativo ya que no se encuentra registrado en el inventario de aplicaciones ni fue posible obtener documentación sobre la solución tecnológica.

* 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**

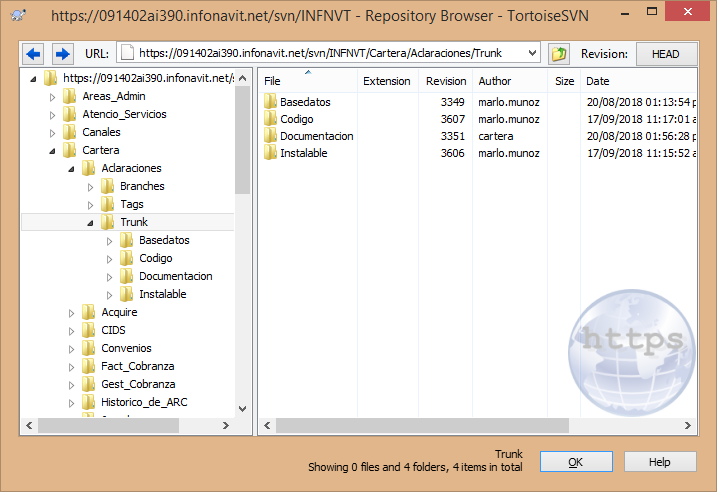
Los errores categorizados anteriormente son clasificados por severidades las cuales son definidas con base en 2 factores: **Impacto** y **Probabilidad**.

|  | **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/*](https://091402ai390.infonavit.net/svn/INFNVT/xxxxxxx)Cartera/Aclaraciones/Trunk



* + 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.aclaraciones

sonar.projectName=Aclaraciones

sonar.projectVersion=1.0

sonar.scm.disabled=True

#Sources

sonar.sources=.

#Bin

sonar.java.binaries=scc

#Exclusions

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

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



* + 1. Ejecución de Análisis de Código Fuente

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



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



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

* + 1. Resultado del Análisis de Código Fuente

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

Ilustración 7 Gráfico de *Bug*s

Ilustración 8 Gráfico de Vulnerabilidades

Ilustración 9 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. | 30 |
| 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. | 11 |
| 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. | 10 |
| Bug | *Major* | **Las asignaciones no deben ser redundantes**  La propiedad transitiva indica que si A == B y B == C, entonces A == C. En tales casos, no es necesario asignar A a C o viceversa porque ya son equivalentes. Esta regla plantea un problema cuando una asignación es inútil porque la variable asignada ya contiene el valor en todos los trayectos de ejecución. | 3 |
| 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. | 2 |
| Bug | *Major* | **No se deben hacer verificaciones de igualdad**  Las comparaciones de diferentes tipos siempre dan como resultado *false*. Por lo que las comparaciones y el código dependiente debe eliminarse. Esto incluye:   * Comparando un objeto con *null.* * Comparando un objeto con un primitivo no emparentado (E. G. *string* con un int). | 2 |
| Bug | *Major* | **Todos los *branches* de una estructura condicional no deberían tener exactamente la misma implementación**  Tener todos los *branches* en un *switch* o en una cadena *if* con la misma implementación es un error. Esta regla no se aplica a las cadenas *if* sin hacer uso de *else* o en caso de utilizar *switch* sin cláusulas por defecto. | 2 |
| Bug | *Major* | **No se deben ignorar los valores de retorno de las funciones sin efectos secundarios**  Cuando la llamada a una función no tiene efectos secundarios. En tal caso, o bien la llamada a la función no es viable y debe ser eliminada o el código fuente no se comporta como se esperaba.  Para evitar la generación de falsos positivos, esta regla activa una incidencia sólo en la siguiente lista predefinida de clases inmutables en la *API* de *Java*:   * java.lang.String * java.lang.booleano * Entero java.lang. * java.lang.Doble * java.lang.float * java.lang.byte * java.lang.Character * java.lang.Short * java.lang.stackTraceElement * java.time.DayOfWeek * java.time.Duration * java.time.Instant * java.time.LocalDate * java.time.LocalDateTime * java.time.LocalTime * java.time.Month * java.time.MonthDay * java.time.OffsetDateTime * java.time.OffsetTime * java.time.Period * java.time.Year * java.time.yearMonth * java.time.ZonedDateTime * java.math.BigInteger * java.math.BigDecimal   java.util.Optional | 1 |
| Bug | *Minor* | **El *boxing* y *unboxing* no debe revertirse inmediatamente**  *Boxing* es el proceso de poner un valor primitivo en un objeto análogo, como por ejemplo crear un entero para mantener un valor. El *Unboxing* es el proceso de recuperar el valor primitivo de tal objeto.    Dado que el valor original no se modifica durante el *boxing* y el *unboxing*, no tiene sentido hacerlo cuando no es necesario. Esto también se aplica al *autoboxing* y *auto*-*unboxing.* | 4 |
| Bug | *Minor* | **Los parámetros del método, las excepciones capturadas y los valores iniciales de las variables *foreach* no deben ignorarse**  Técnicamente correcto asignar a los parámetros desde dentro de los cuerpos del método, hacerlo antes de que se lea el valor del parámetro es considerado un error. En cambio, los valores iniciales de los parámetros, las excepciones capturadas y los parámetros de *foreach* deben ser tratados como finales o leídos antes de la reasignación. | 2 |
| Bug | *Minor* | **Los resultados de "*compareTo*" no deben verificarse para valores específicos**  Mientras que la mayoría de los métodos de *comparaciónTo* devuelven -1,0 o 1, algunos no lo hacen y probar el resultado de *compareTo* contra un valor específico distinto de 0 podría dar lugar a falsos negativos. | 1 |
| Vulnerabilidad | *Critical* | **Deben utilizarse filtros definidos**  Cada filtro definido en el archivo *web.xml* debe ser usado en un elemento *<filter-mapping>*. De lo contrario, no se invocarán estos filtros. | 1 |
| Vulnerabilidad | *Minor* | **Las direcciones *IP* no deben estar en código duro**  Poner en duro dirección *IP* dentro del código fuente es una mala idea por varias razones:   * Se necesita compilar de nuevo si la dirección *IP* cambia. * Obliga a utilizar la misma dirección en todos los ambientes (Desarrollo, Pruebas y Producción). * Deja la responsabilidad de colocar el valor a utilizar en producción en manos del desarrollador. * Permite a los atacantes descompilar el código y descubrir direcciones potencialmente sensibles. | 1 |
| *Optimizador* | *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.* | 1 |
| Optimizador | *Critical* | **Las literales de los *strings* no deben duplicarse**  Las cadenas de texto duplicadas literalmente hacen que el proceso de refactorización sea propenso a errores, ya que debe asegurarse de actualizar todas las ocurrencias. Por otra parte, las constantes pueden ser referenciadas desde muchos lugares, pero sólo necesitan ser actualizadas en un solo lugar. El umbral por defecto es de 3. | 76 |
| 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. | 43 |
| 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. | 31 |
| 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 | 23 |
| Optimizador | *Critical* | **Se debe utilizar *try-with-resources***  *Java 7* introdujo la declaración *try-with-resources*, que garantiza el cierre del recurso en cuestión. Esta regla verifica que los recursos que puedan cerrarse se abren en una declaración de prueba con recursos. | 22 |
| Optimizador | *Major* | **Los archivos fuente no deben tener bloques duplicados**  Los archivos fuente no deben tener bloques duplicados. | 113 |
| 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. | 37 |
| 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. | 30 |
| 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. | 28 |
| 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. | 27 |
| 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. | 27 |
| 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. | 21 |
| 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* | **Las clases sincronizadas *Vector*, *Hashtable*, *Stack* y *StringBuffer* no se deben utilizar**  Las primeras clases de la *API* de *Java,* como *Vector*, *Hashtable* y *StringBuffer*, se sincronizaron para hacerlas seguras para los hilos. La sincronización tiene un gran impacto negativo en el rendimiento, incluso cuando se utilizan estas colecciones desde un solo hilo. Es mejor usar sus nuevos reemplazos no sincronizados:   * *ArrayList* o *LinkedList* en lugar de *Vector.* * *Deque* en lugar de *Stack.* * *HashMap* en lugar de *Hashtable.*   *StringBuilder* en lugar de *StringBuffer* | 10 |
| Optimizador | *Major* | **Las sentencias "*if*" colapsables deberían fusionarse**  Fusionar sentencias *if* colapsables incrementan la legibilidad del código. | 7 |
| Optimizador | *Major* | **Las excepciones genéricas nunca deben ser lanzadas**  El uso de excepciones genéricas como *Error, RuntimeException, Throwable* y *Exception* evita que los métodos de llamada manejen las excepciones reales generadas por el sistema de manera diferente a los errores generados por la aplicación. | 5 |
| Optimizador | *Major* | **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 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* | ***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* | **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 |

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 10 Resultado General - Código

Ilustración 11 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 12 Gráfico Comparativo - Código

Ilustración 13 Comparativo – Incidencias General

Ilustración 14 Gráfico Comparativo de *Bug*s

Ilustración 15 Gráfico Comparativo de Vulnerabilidades

Ilustración 16 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. 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.