TechForge

Informe de análisis estático

Santiago Bejarano
santiago.bejarano@epn.edu.ec

Andrés Cantuña
edwin.cantuna@epn.edu.ec

Isaac Friedman
isaac.friedman@epn.edu.ec

Matías Mejía
matias.mejia@epn.edu.ec

Alexis Lapo
alexis.lapo@epn.edu.ec

3 de Agosto de 2025

Índice general

1 Informe Ejecutivo

1.1. Introducción

El presente informe presenta los hallazgos identificados en el análisis estático de código realizado con **SonarQube** sobre el proyecto **Core Bancario Python**, con fecha de análisis `2025-08-03`.

El objetivo del análisis fue detectar automáticamente errores, malas prácticas y problemas de mantenibilidad a nivel del código fuente. El enfoque fue exclusivamente sobre la revisión sintáctica y semántica del código, sin realizar ejecución ni pruebas dinámicas.

El análisis abarcó aspectos como:

- Mantenibilidad del código (Code Smells)
- Fiabilidad y potenciales bugs
- Adherencia a buenas prácticas de programación

1.2. Metodología

El análisis se realizó utilizando **SonarQube** como motor de análisis estático. Se emplearon reglas específicas para Python (por ejemplo, `python:S5843`, `python:S1192`, `python:S1066`, `python:S3776`, etc.) para evaluar la calidad y seguridad del código fuente.

Los hallazgos se agruparon por tipo y severidad, considerando una entrada por cada regla infringida para mayor claridad, aunque algunas reglas presentaron múltiples ocurrencias en diferentes archivos.

1.3. Resumen Ejecutivo

El análisis identificó un total de **9 tipos de errores únicos**, clasificados en su totalidad como **Code Smells**, que afectan la calidad y mantenibilidad del código. No se detectaron Bugs o Vulnerabilidades directas, pero los problemas identificados podrían ocultar errores lógicos o dificultar futuras modificaciones.

Se identificaron debilidades en las siguientes áreas:

- **Expresiones Regulares Ineficientes:** Uso de patrones con caracteres duplicados.
- **Duplicación de Código:** Literales de texto (mensajes de error) y consultas SQL repetidas.
- **Complejidad del Código:** Funciones con alta complejidad cognitiva y uso de ‘if‘ anidados innecesarios.

- **Prácticas de Código Limpio:** Presencia de código comentado y uso incorrecto de f-strings.

La siguiente tabla resume los hallazgos principales:

Tipo	Cantidad de reglas	Ocurrencias totales	Lenguajes afectados
Code Smell	9	25	Python

Cuadro 1.1: Resumen de hallazgos por tipo

La distribución de errores por severidad es la siguiente:

- **Critical:** 4 reglas
- **Major:** 4 reglas
- **Minor:** 1 regla

El análisis revela que el archivo `app/main.py` concentra la mayor cantidad de problemas, seguido por `app/validators.py`, lo que sugiere que estos archivos podrían ser candidatos prioritarios para refactorización.

2 Informe Técnico

2.1. Alcance y Activos Evaluados

Durante el análisis estático de código realizado sobre el proyecto **Core Bancario Python**, se evaluaron los siguientes archivos críticos que constituyen los componentes principales del sistema:

- `app/main.py` (Lógica principal de la aplicación bancaria)
- `app/validators.py` (Validadores y funciones de seguridad)
- `app/custom_logger.py` (Sistema de logging y enmascaramiento de datos)

Estos archivos representan las funcionalidades más críticas del sistema bancario, incluyendo operaciones financieras, validaciones de seguridad, gestión de cuentas y tarjetas de crédito, y el sistema de auditoría mediante logs.

2.2. Hallazgos de calidad y mantenibilidad (Code Smells)

2.2.1. S5843: Expresiones regulares ineficientes

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S5843	Code Smell	Las expresiones regulares deben ser eficientes. Se detectaron clases de caracteres con duplicados que reducen el rendimiento.	Python	5

Cuadro 2.1: Expresiones regulares ineficientes

Detalle: Se identificaron 5 instancias de expresiones regulares que contienen clases de caracteres con duplicados en el sistema de enmascaramiento de logs, afectando el rendimiento del sistema de auditoría.

Recomendación: Optimizar las expresiones regulares eliminando caracteres duplicados en las clases de caracteres para mejorar el rendimiento del sistema de logging.

Ejemplo representativo:

```
# Problemático:
r"([a-zA-Z0-9_.+-]+)@([a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)"
# Optimizado:
r"([a-zA-Z0-9_.+-]+)@([a-zA-Z0-9.-]+\.[a-zA-Z0-9.-]+)"
```

2.2.2. S1192: Literal .El monto debe ser mayor que cero"duplicado

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S1192	Code Smell	Los literales de texto no deben estar duplicados. El mensaje .El monto debe ser mayor que cero"se repite 4 veces.	Python	4

Cuadro 2.2: Literal .El monto debe ser mayor que cero"duplicado

Detalle: El mensaje de error .El monto debe ser mayor que cero.está duplicado 4 veces en el código, violando el principio DRY y dificultando el mantenimiento de validaciones de montos.

Recomendación: Definir una constante para este mensaje de error específico y utilizarla en todas las validaciones de montos del sistema.

Ejemplo representativo:

```
# Problemático:
api.abort(400, "El monto debe ser mayor que cero")

# Recomendado:
INVALID_AMOUNT_MESSAGE = "El monto debe ser mayor que cero"
api.abort(400, INVALID_AMOUNT_MESSAGE)
```

2.2.3. S1192: Literal Çuenta no encontrada"duplicado

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S1192	Code Smell	Los literales de texto no deben estar duplicados. El mensaje Çuenta no encontrada"se repite 3 veces.	Python	3

Cuadro 2.3: Literal Çuenta no encontrada"duplicado

Detalle: El mensaje de error Çuenta no encontrada.está duplicado 3 veces en el código, afectando la consistencia en las respuestas de error de operaciones bancarias.

Recomendación: Centralizar este mensaje crítico en una constante para asegurar consistencia en todas las operaciones que requieren validación de existencia de cuentas.

Ejemplo representativo:

```
# Problemático:
api.abort(404, "Cuenta no encontrada")

# Recomendado:
ACCOUNT_NOT_FOUND_MESSAGE = "Cuenta no encontrada"
api.abort(404, ACCOUNT_NOT_FOUND_MESSAGE)
```

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S1192	Code Smell	Los literales de texto no deben estar duplicados. La consulta SQL se repite 5 veces en el código.	Python	5

Cuadro 2.4: Consulta SQL duplicada

2.2.4. S1192: Consulta SQL duplicada

Detalle: La consulta SQL "SELECT balance FROM bank.accounts WHERE user$_id$ =

Recomendación: Definir la consulta SQL en una constante para centralizar el acceso a datos de balance y facilitar futuras modificaciones del esquema.

Ejemplo representativo:

```
# Problemático:
cur.execute("SELECT balance FROM bank.accounts WHERE user_id = %s", (user_id,))

# Recomendado:
GET_BALANCE_QUERY = "SELECT balance FROM bank.accounts WHERE user_id = %s"
cur.execute(GET_BALANCE_QUERY, (user_id,))
```

2.2.5. S1192: Literal "Tarjeta de crédito no encontrada"duplicado

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S1192	Code Smell	Los literales de texto no deben estar duplicados. El mensaje "Tarjeta de crédito no encontrada"se repite 4 veces.	Python	4

Cuadro 2.5: Literal "Tarjeta de crédito no encontrada"duplicado

Detalle: El mensaje de error "Tarjeta de crédito no encontrada.ªtá duplicado 4 veces en el código, afectando tanto las respuestas de API como los logs del sistema.

Recomendación: Consolidar este mensaje en una constante para mantener consistencia en las operaciones de tarjetas de crédito y facilitar auditorías.

Ejemplo representativo:

```
# Problemático:
log_event('ERROR', "Tarjeta de crédito no encontrada")
api.abort(404, "Tarjeta de crédito no encontrada")

# Recomendado:
CREDIT_CARD_NOT_FOUND_MESSAGE = "Tarjeta de crédito no encontrada"
log_event('ERROR', CREDIT_CARD_NOT_FOUND_MESSAGE)
api.abort(404, CREDIT_CARD_NOT_FOUND_MESSAGE)
```

2.2.6. S3776: Complejidad cognitiva excesiva en funciones

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S3776	Code Smell	La complejidad cognitiva de las funciones no debe ser demasiado alta. Funciones muy complejas dificultan la comprensión y el mantenimiento del código.	Python	1

Cuadro 2.6: Complejidad cognitiva excesiva en funciones

Detalle: Se identificó la función `validar_password` con una complejidad cognitiva de 16, superando el límite recomendado de 15. Esta función crítica para la seguridad del sistema maneja múltiples validaciones en una sola función.

Recomendación: Refactorizar la función dividiendo cada validación en subfunciones más específicas, aplicando el principio de responsabilidad única.

Ejemplo representativo:

```
# Refactorización recomendada:
def validar_password(password: str, info_personal: dict) -> bool:
    validaciones = [
        _validar_longitud,
        _validar_mayusculas,
        _validar_minusculas,
        _validar_numeros
    ]
    return all(v(password) for v in validaciones)
```

2.2.7. S1066: Sentencias if anidadas fusionables

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S1066	Code Smell	Las sentencias ïf"fusionables deben combinarse para reducir la complejidad del código.	Python	1

Cuadro 2.7: Sentencias if anidadas fusionables

Detalle: Se detectaron sentencias `if` anidadas innecesarias en la validación de nombres de usuario que pueden ser combinadas usando el operador lógico `and`.

Recomendación: Fusionar las condiciones `if` anidadas en una sola sentencia para mejorar la legibilidad y reducir la complejidad ciclomática.

Ejemplo representativo:

```
# Problemático:
if len(parte) > 4:
```

```
    if username_lower == parte:
        return False

# Recomendado:
if len(parte) > 4 and username_lower == parte:
    return False
```

2.2.8. S125: Código comentado debe eliminarse

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S125	Code Smell	Las secciones de código no deben estar çomentadas". El código muerto reduce la legibilidad.	Python	1

Cuadro 2.8: Código comentado debe eliminarse

Detalle: Se detectó código comentado innecesario en la configuración del sistema de logging que ya no tiene utilidad y puede generar confusión.

Recomendación: Eliminar todas las líneas de código comentadas que no aporten valor documental al proyecto.

Ejemplo representativo:

```
# Línea a eliminar:
#log = logging.getLogger(__name__)
```

2.2.9. S3457: F-strings mal utilizados

Regla	Tipo	Descripción	Lenguaje	Ocurrencias
S3457	Code Smell	Los strings con formato printf no deben llevar a comportamientos inesperados en tiempo de ejecución.	Python	1

Cuadro 2.9: F-strings mal utilizados

Detalle: Se detectó el uso de una f-string para una cadena de texto que no contiene ninguna expresión a formatear, lo cual es innecesario y puede generar confusión.

Recomendación: Convertir las f-strings que no contienen expresiones a interpolizar en strings normales.

Ejemplo representativo:

```
# Problemático:
log_event('WARN', f"Intento de transferencia a la misma cuenta")

# Recomendado:
log_event('WARN', "Intento de transferencia a la misma cuenta")
```

2.3. Resumen e Informe de Vulnerabilidades

Tras completar el análisis estático del proyecto Core Bancario Python, se presenta a continuación un resumen consolidado de los hallazgos. Esta tabla resume los hallazgos clasificados por severidad, destacando el número total de problemas de calidad y mantenibilidad encontrados durante el proceso de evaluación del código fuente.

2.3.1. Resumen por categoría de hallazgo

La siguiente tabla muestra el número de hallazgos detectados por categoría:

Code Smell	Hotspot de Seguridad	Bug
9	0	0

Cuadro 2.10: Cantidad de hallazgos por categoría

2.3.2. Distribución por severidad

La siguiente tabla muestra la distribución de hallazgos por nivel de severidad:

Severidad	Cantidad de reglas	Ocurrencias totales
Critical	4	13
Major	4	11
Minor	1	1

Cuadro 2.11: Distribución de hallazgos por severidad

2.3.3. Hallazgos críticos del análisis estático

La siguiente tabla proporciona una vista consolidada de los hallazgos más relevantes identificados durante el análisis, priorizando aquellos que requieren atención inmediata por parte del equipo de desarrollo:

Hallazgo	Severidad	Recomendación técnica
SAST-001: Expresiones regulares ineficientes	**Major**	Optimizar patrones regex eliminando caracteres duplicados
SAST-002: Literal .[El] monto debe ser mayor que cero"	**Major**	Crear constante para mensaje de error de montos
SAST-003: Literal Çuenta no encontrada"	**Critical**	Centralizar mensaje de error de cuentas
SAST-004: Consultas SQL duplicadas	**Critical**	Definir constante para consulta de balance
SAST-005: Literal "Tarjeta de crédito no encontrada"	**Critical**	Consolidar mensaje de error de tarjetas
SAST-006: Complejidad cognitiva en validar_password	**Critical**	Refactorizar función en subfunciones más simples
SAST-007: Sentencias if anidadas fusionables	**Major**	Combinar condiciones if usando operador and
SAST-008: F-string mal utilizado	**Major**	Usar string normal cuando no hay interpolación
SAST-009: Código comentado innecesario	**Minor**	Eliminar líneas de código comentadas

Cuadro 2.12: Hallazgos críticos del análisis estático

2.3.4. Distribución por archivos

La siguiente tabla muestra cómo se distribuyen los hallazgos por archivo del proyecto:

Archivo	Cantidad	Tipos de problemas
`app/main.py`	6	Literales duplicados, consultas SQL repetidas, f-string mal usado, código comentado
`app/validators.py`	2	Complejidad cognitiva excesiva, sentencias if anidadas
`app/custom_logger.py`	1	Expresiones regulares ineficientes

Cuadro 2.13: Distribución de hallazgos por archivo

2.3.5. Recomendaciones prioritarias

Basado en el análisis realizado, se recomienda priorizar las siguientes acciones correctivas:

1. **Refactorización de la función validar_password:** Por su criticidad en la seguridad del sistema
2. **Centralización de mensajes de error:** Para mejorar la consistencia y facilitar internacionalización
3. **Optimización del sistema de logging:** Para mejorar el rendimiento del sistema de auditoría

4. **Consolidación de consultas SQL:** Para reducir la duplicación de código y facilitar mantenimiento