UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

La Universidad Católica de Loja

ÁREA TÉCNICA

TÍTULO DE INGENIERO EN SISTEMÁS INFORMÁTICOS Y COMPUTACIÓN

Identificación de Technical debt (TD) en aplicaciones de software construidas bajo estilos y patrones arquitectónicos, a partir del análisis del código fuente o ejecutable.

TRABAJO DE TITULACIÓN.

AUTOR: Rosillo Reyes, Juan Pablo

DIRECTOR: Guamán Coronel, Daniel Alejandro

LOJA – ECUADOR

2017

Esta versión digital, ha sido acreditada bajo la licencia Creative Commons 4.0, CC BY-NY-SA: Reconocimiento-No comercial-Compartir igual; la cual permite copiar, distribuir y comunicar públicamente la obra, mientras se reconozca la autoría original, no se utilice con fines comerciales y se permiten obras derivadas, siempre que mantenga la misma licencia al ser divulgada. http://creativecommons.org/licenses/by-nc-sa/4.0/deed.es

Septiembre, 2017

APROBACIÓN DEL DIRECTOR DEL TRABAJO DE TITULACIÓN

Ingeniero
Daniel Alejandro Guamán Coronel
DOCENTE DE LA TITULACIÓN

De mi consideración:

El presente trabajo de titulación: "Identificación de Technical debt (TD) en aplicaciones de software construidas bajo estilos y patrones arquitectónicos, a partir del análisis del código fuente o ejecutable" realizado por el señor estudiante Rosillo Reyes Juan Pablo, ha sido orientado y revisado durante su ejecución, por cuanto se aprueba la presentación del mismo.

Loja, febrero de 2017

f)………………………..

DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS

Yo, Rosillo Reyes Juan Pablo declaro ser autor del presente trabajo de titulación: Identificación de Technical debt (TD) en aplicaciones de software construidas bajo estilos y patrones arquitectónicos, a partir del análisis del código fuente o ejecutable, de la Titulación de Sistemas Informáticos y Computación, siendo Daniel Alejandro Guamán Coronel director del presente trabajo; y eximo expresamente a la Universidad Técnica Particular de Loja y a sus representantes legales de posibles reclamos o acciones legales. Además certifico que las ideas, conceptos, procedimientos y resultados vertidos en el presente trabajo investigativo, son de mi exclusiva responsabilidad.

Adicionalmente declaro conocer y aceptar la disposición del Art. 88 del Estatuto Orgánico de la Universidad Técnica Particular de Loja que en su parte pertinente textualmente dice: “Forman parte del patrimonio de la Universidad la propiedad intelectual de investigaciones, trabajos científicos o técnicos y tesis de grado que se realicen a través, o con el apoyo financiero, académico o institucional (operativo) de la Universidad.

f……………………………
Juan Pablo Rosillo Reyes
1105027195

DEDICATORIA

La presente Tesis está dedicada a
Dios, y a mis Padres Manuel Rosillo
y Mercedes Reyes, ya que han sido
mi inspiración para cumplir mis objetivos.

Juan Pablo

AGRADECIMIENTO

Quiero expresar mi sincero agradecimiento a la Universidad Técnica Particular de Loja, Institución que ha sabido labrarse un espacio de gloria en nuestra ciudad, así como a nivel nacional e internacional.

A los catedráticos, quienes a través de las enseñanzas impartidas supieron formarme académicamente.

Así mismo dejo constancia de nuestra especial gratitud a la titulación de Sistemas Informáticos y Computación, en persona de sus mentalizadores y al Director de Tesis, por su apoyo relevante a la cristalización de mi aspiración académica y al desarrollo de un pensamiento investigativo.

INDICE DE CONTENIDOS

APROBACIÓN DEL DIRECTOR DEL TRABAJO DE TITULACIÓN ii
DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS iii
DEDICATORIA iv
AGRADECIMIENTO v
INDICE DE CONTENIDOS vi
INDICE DE TABLAS viii
INDICE DE FIGURAS ix
INDICE DE ECUACIONES x
INDICE DE ANEXOS xi
RESUMEN 1
ABSTRACT 2
INTRODUCCIÓN 3
Objetivo General 4
Objetivos Específicos 4
Metodología para el desarrollo del trabajo 5
1. CAPITULO 1: MARCO TEÓRICO 6
1.1 Deuda Técnica 7
1.1.1 Clasificación de la deuda técnica. 8
1.1.2 Tipos de deuda técnica. 9
1.1.2.1 Deuda Técnica de Código. 10
1.1.2.2 Deuda Técnica de Diseño. 10
1.1.3 Causas de la acumulación de la deuda técnica Arquitectónica. 11
1.2 Gestión de la Deuda Técnica. 12
1.2.1 Actividades de la gestión de la deuda técnica. 12
1.2.1.1 Identificación de la deuda técnica. 13
1.2.1.2 Medición de la deuda técnica. 14
1.2.1.3 Monitoreo de la deuda técnica. 14
1.3 Análisis estático. 15
1.4 Code Smells. 17
1.5 Anti patrones de diseño y acumulación de suciedad. 18
1.6 Violaciones Modulares. 19
1.7 Herramientas para la identificación de deuda técnica. 20
1.8 Modelos de calidad para la identificación de deuda técnica. 36
2. CAPITULO 2: PROCESO DE IDENTIFICACIÓN Y MEDICIÓN DE LA DEUDA TÉCNICA 39
2.1 Identificación y medición de la deuda técnica. 40
2.1.1 Método SQALE. 44
2.1.1.1 Principios de SQALE. 46
2.1.1.2 Estructura SQALE. 47
2.1.1.3 Modelo de calidad 47
2.1.1.4 Modelo de análisis 47
2.1.1.5 Los índices 49

2.1.1.6 Los indicadores 51
2.2 Medidas de cálculo de la deuda técnica........ 53
2.2.1 Características SQALE a evaluar. 55
3. CAPITULO 3: IDENTIFICACIÓN Y MEDICIÓN DE LA DEUDA TÉCNICA........ 57
3.1 Herramientas para el análisis de calidad de software. 58
3.1.1 SonarQube........ 59
3.1.2 Kiuwan. 61
3.1.3 PMD. 62
3.1.4 LocMetrics........ 63
3.2 Análisis estático de software a través de herramientas de calidad de software........ 64
3.2.1 Aplicaciones de software a evaluar. 64
3.2.2 Configuración para uso de herramientas........ 65
3.2.3 Análisis e interpretación de resultados de medición de la Deuda Técnica........ 67
3.2.3.1 Fase de Identificación. 67
3.2.3.2 Fase de Medición. 69
3.2.3.3 Fase de Monitoreo. 74
3.2.4 Otras herramientas........ 81
CONCLUSIONES 85
RECOMENDACIONES 89
ANEXOS........ 90
BIBLIOGRAFÍA........ 130

INDICE DE TABLAS

Tabla1. Ventajas y desventajas del análisis estático. 16
Tabla 2. Ventajas y desventajas de Code Smells. 18
Tabla 3. Ventajas y desventajas de la detección de suciedad en patrones de diseño. 19
Tabla 4. Ventajas y desventajas de la detección de violaciones modulares. 20
Tabla 5. Herramientas para la gestión de la deuda técnica. 21
Tabla 6. Tabla de costos de remediación 47
Tabla 7. Tabla de costos de no remediación 48
Tabla 8. Índices consolidados 50
Tabla 9. Escala de categorización del ratio SQALE 52
Tabla 10.Ejemplo de características y subcaracterísticas de método SQALE. 53
Tabla 11. Selección de subcaracterísticas y reglas para el análisis de deuda técnica. 55
Tabla 12. Evaluación de herramientas con respecto a sus métricas. 58
Tabla 13.Datos de entrada y salida de SonarQube. 60
Tabla 14. Datos de entrada y salida de Kiuwan 62
Tabla 15. Datos de entrada y salida de PMD 63
Tabla 16 Entrada y salida de datos de la herramienta LocMetrics 64
Tabla 17.Soluciones de software para el análisis de calidad. 64
Tabla 18.Resultado de análisis de deuda técnica por herramienta. 69

INDICE DE FIGURAS

Figura 1.Actividades de la gestión de la deuda técnica ... 12
Figura 2.Interfaz de usuario de la herramienta CAST AIP. ... 22
Figura 3.Interfaz de usuario de SonarQube. ... 23
Figura 4.Ejemplo de análisis con plugin FindBugs en NetBeans. ... 24
Figura 5. Interfaz de usuario de Structure 101. ... 25
Figura 6.Ejemplo de análisis de métricas con Understand. ... 26
Figura 7.Ejemplo de visualización de dependencias con SonarGraph. ... 27
Figura 8. Ejemplo de análisis de código con SourceMeter. ... 28
Figura 9. Interfaz de usuario de Ndepend. ... 29
Figura 10. Ejemplo de reporte de análisis de calidad de software de Lattix. ... 30
Figura 11. Ejemplo de funcionamiento de herramienta SourceMonitor. ... 31
Figura 12.Dashboard de herramienta SQUORE. ... 32
Figura 13.Dashboard de herramienta Kiuwan. ... 33
Figura 14. Ejemplo de reporte de análisis de software presentado por PMD ... 34
Figura 15. Proceso de análisis de calidad de software con Mia-Quality. ... 35
Figura 16.Proceso de identificación de deuda técnica a nivel de código. ... 41
Figura 17.Proceso de identificación y medición de deuda técnica a nivel de arquitectura. ... 42
Figura 18.Proceso de Gestión de deuda técnica. ... 43
Figura 19.Características de calidad de SQALE. ... 45
Figura 20.Calificación de SQALE ... 51
Figura 21. Resumen de da DT de un proyecto ... 52
Figura 22.Ejemplo de pirámide SQALE de la herramienta SonarQube. ... 53
Figura 23. Pantalla de control de SonarQube ... 59
Figura 24.Resumen de la deuda técnica de un proyecto ... 60
Figura 25.Widgets de Kiuwan sobre la deuda técnica. ... 61
Figura 26.Ejemplo de reporte generado por la herramienta PMD. ... 62
Figura 27. Ejemplo de reporte generado por la herramienta LocMetrics ... 63
Figura 28. Deuda técnica por herramienta de las aplicaciones 7 y 8 ... 71
Figura 29. Promedio de deuda técnica por característica de todas las aplicaciones. ... 72
Figura 30.Pirámide SQALE del primer análisis de DT de aplicación Analizador Léxico. ... 73
Figura 31. Métricas con mayor influencia en aplicaciones. ... 74
Figura 32. Variación de tamaño de aplicaciones 6 y 8 tras aplicar cambios para disminuir la deuda técnica. ... 75
Figura 33. Número de violaciones por herramienta. ... 76
Figura 34. Variación de número de violaciones de comentarios requeridos. ... 76
Figura 35. Complejidad ciclomática de las aplicaciones 5 y 8. ... 77
Figura 36. Número de clases con problemas de acoplamiento de las aplicaciones 7 y 8. ... 78
Figura 37. Resultado de duplicaciones de aplicación 8. ... 79
Figura 38.Resultados de código duplicado de Analizador Léxico con SonarQube. ... 80
Figura 39. Calificación SQALE de aplicaciones del primer análisis de DT. ... 81
Figura 40. Análisis de diseño con Jarchitect para la App 7 ... 82
Figura 41. Dependencias de clases presentada por Jarchitect de la App7 ... 83
Figura 42. Resultado de Análisis de la App 7 por medio de SourceMonitor ... 83
Figura 43. Resultado de forma gráfica de Análisis de la App 7 por medio de SourceMonitor ... 84

INDICE DE ECUACIONES

(Ecuación 1). Valor de deuda técnica por característica 50
(Ecuación 2). Valor total de deuda técnica. 50
(Ecuación 3). Índice Comprobabilidad 50
(Ecuación 4). Índice Consolidado Confiabilidad 50
(Ecuación 5). Índice Consolidad Mantenibilidad 50
(Ecuación 6). Índice Capacidad de Cambio 50
(Ecuación 7). Ratio de la deuda técnica 52

INDICE DE ANEXOS

Anexos 1. Glosario de términos 91
Anexos 2. Reglas de código para el análisis de deuda técnica en lenguaje JAVA 92
Anexos 3. Reglas de código de la herramienta Kiuwan para el análisis de deuda técnica en lenguajes JAVA Y PHP 94
Anexos 4. Reglas de código de la herramienta PMD para el análisis de deuda técnica en lenguaje JAVA 102
Anexos 5. Resultados de análisis por característica con la herramienta SonarQube 103
Anexos 6. Resultados de análisis por característica con la herramienta Kiuwan................ 112
Anexos 7. Resultados de análisis por característica con la herramienta PMD 123
Anexos 8. : Violaciones de código a corregir.. 125

RESUMEN

El presente trabajo de titulación tiene como finalidad aplicar procesos y actividades para la identificación de deuda técnica en aplicaciones de software, las mismas que en algunos casos son construidas siguiendo una arquitectura de software, pero que debido a defectos en su diseño y uso de patrones contienen fallos que luego se ven reflejados en su codificación; éstos suelen acumularse haciendo que su mantenimiento, rendimiento y robustez se vea afectada. Para ello se realiza un análisis previo sobre la deuda técnica, las causas que conducen hacia ella y su influencia en el código fuente y diseño de las aplicaciones; además se identificaron herramientas que permiten automatizar los temas de deuda técnica, y métodos como el SQALE que apoyan en ésta tarea. Posteriormente se procedió a la validación de las herramientas utilizando como técnica de análisis el estático, el mismo que toma como entrada el código fuente escrito en distintos lenguajes de programación, para identificar la existencia en un alto, medio o bajo grado de deuda y realizar las correcciones necesarias para su reducción, facilitando cualquier cambio que requieran las aplicaciones.

PALABRAS CLAVES: deuda técnica, deuda técnica de código, deuda técnica arquitectónica, análisis estático.

ABSTRACT

The purpose of the present titling work is to apply processes and activities for the identification of technical debt in software applications, which in some cases are built following a software architecture, but due to defects in their design and use of standards contain Faults that are then reflected in their coding; These usually accumulate causing their maintenance, performance and robustness to be affected. For this, a prior analysis is made on the technical debt, the causes that lead to it and its influence on the source code and the design of the applications; In addition, tools were identified that allow automating technical debt issues, and methods such as SQALE that support this task. Subsequently, the validation of the tools was performed using the static analysis technique, which takes as input the source code written in different programming languages, to identify the existence in a high, medium or low degree of debt and perform the Corrections necessary for its reduction, facilitating any changes that the applications require.

KEY WORDS: technical debt, code technical debt, architectural technical debt, static analysis.

INTRODUCCIÓN

Un producto software posee características asociadas que garantizan la calidad del mismo, como: un lenguaje de programación adecuado, el uso de estándares de calidad y la implementación de una arquitectura soportada por estilos y patrones arquitectónicos. La calidad, parte primordial en el diseño y codificación garantiza la satisfacción completa de las necesidades del usuario final; además mejora la economía de las organizaciones debido a que un producto bien elaborado reduce los costos de mantenimiento y facilita su evolución y mantenibilidad.

Sin embargo, debido a los ajustados tiempos de construcción del software y la aplicación de soluciones sub-óptimas se genera un producto final en mal estado, que considera gastos los mismos que pueden ser muy costosos si se detectan problemas derivados de imperfecciones en el diseño, por lo que resulta imprescindible analizar el estado del producto e identificar cual es el coste necesario para mejorar la calidad del mismo.

Los equipos de desarrollo de software, son los encargados de cuantificar y expresar en términos de negocio el costo o esfuerzo que se requiere para corregir los errores de software, lo cual se conoce como deuda técnica. La deuda técnica es un término introducido por (Cunningham, 1992), que representa un interés monetario que se agrega como un coste adicional del presupuesto de desarrollo, para la mejora del producto de software, el cual que debe ser pagado por el proveedor que desarrolla el software.

Para poder identificar la deuda técnica, se puede ejecutar un análisis estático o dinámico en las aplicaciones que posean errores de código y/o diseño, y se lo puede hacer a través de herramientas de evaluación de la calidad del software, que se enfocan a la detección de este problema; además, es necesario aplicar normas y métodos, como: ISO / IEC 25010, ISO /IEC 9126 y SQALE, las cuales proveen de métricas y criterios de evaluación que se utilizan para estimar la calidad e identificar la deuda técnica en el código fuente de las aplicaciones.

El presente trabajo tiene como objetivo, cuantificar la deuda técnica, a través de herramientas y metodologías que faciliten los procesos de identificación y análisis, tomando en cuenta características relacionados al código y al diseño de una aplicación, como: estilos arquitectónico, patrón arquitectónico, patrón de diseño y atributos de calidad con los que se desarrolla un producto de software; para ello se utilizará el análisis estático tomando como

elemento de entrada el código fuente de las aplicaciones de software y con ello cuantificar la existencia de deuda técnica.

El presente documento contiene la siguiente estructura:

El capítulo I, describe un marco teórico, en el que detalla las definiciones esenciales de la deuda técnica y de herramientas encontradas tras un estudio de las mismas.

El capítulo II describe el proceso de la identificación de la deuda técnica y que consideraciones de deben tomar en cuenta para la ejecución del mismo.

En el capítulo III describe y ejecuta las herramientas de análisis estático para automatizar el proceso de identificación de la DT, aplicándolas en casos que contengan posibles problemas de acumulación de deuda técnica, validando con ello la eficiencia de las mismas, y documentando los resultados obtenidos.

Posteriormente se aborda las conclusiones y recomendación del trabajo realizado.

Finalmente, estos capítulos se desarrollan para dar alcance a los siguientes objetivos:

Objetivo General

Clasificar y comparar herramientas o frameworks que puedan ser utilizadas para identificar la deuda técnica de una solución de software a partir de su código fuente o ejecutable.

Objetivos Específicos

- Identificar, clasificar y comparar herramientas o frameworks comerciales y no comerciales.
- Seleccionar y clasificar soluciones software codificadas con distintos lenguajes de programación para validar las herramientas o frameworks.
- Evaluar y validar herramientas mediante las soluciones de software seleccionadas.
- Documentar resultados obtenidos a partir de la validación de resultados, exponiendo ventajas, desventajas y sugerencias de su uso y aplicabilidad.

Metodología para el desarrollo del trabajo

El presente trabajo empieza por la revisión de conceptos teóricos que son de apoyo para el conocimiento de temas relacionados a la arquitectura y desarrollo de software, a partir de ello se procederá a buscar algunos trabajos similares que hagan referencia a términos o palabras claves como: Technical Debt, Source Code, Tools, architectural technical debt, y con ellos identificar conceptos que ayudarán a responder preguntas tales como: ¿por qué?, ¿cómo? y ¿para qué se utiliza la deuda técnica?.

En el proceso de análisis de herramientas o frameworks, se deben revisar las características, describir sus ventajas y desventajas, clasificarlas con el fin de que puedan servir en el proceso de validación, tomando como referencia que se parte del código fuente o ejecutable el mismo que puede estar codificado en cualquier lenguaje de programación, por ende, el análisis de las herramientas permitirá identificar qué está permitido realizar y qué atributos de arquitectura de software podría extraer.

Para el proceso de validación de las herramientas o frameworks se utilizarán soluciones software desarrolladas por estudiantes de UTPL y se utilizará el análisis estático con el apoyo de las herramientas de análisis de calidad de software, identificar características como: duplicaciones, tamaño de código, complejidad ciclomática, violaciones de diseño y dependencias de clases.

Finalmente con la validación de las soluciones software en herramientas o frameworks se van a obtener resultados, los mismos que deben ser evaluados y clasificados ya que, pueden servir para futuros trabajos dentro de la ingeniería, desarrollo y arquitectura de software.

CAPITULO 1: MARCO TEÓRICO

1.1 Deuda Técnica

La Deuda Técnica (DT) es una metáfora que aborda las consecuencias de un desarrollo pobre de software (Cunningham, 1992), el cual puede volverse insostenible si no se le otorga la suficiente atención. En un ámbito financiero, se define a la deuda técnica como los costes a futuro que son atribuibles a violaciones conocidas en el código de producción que deben ser corregidos; un coste que incluye intereses, los mismos que representan el costo que se paga día a día por mantener la deuda y se suma al costo total de eliminarla . (Curtis, Sappidi, & Szynkarski, 2012a).

Sin embargo, aunque la deuda técnica implica problemas a nivel de software y de organización, se puede ver también como un beneficio. (Brown et al., 2010) define a la deuda como, "Una situación en la que se negocia la calidad del código a largo plazo para obtener beneficios a corto plazo", es decir, se puede incluir siempre y cuando se tome en cuenta su debido control, tal como lo menciona el autor (Allman, 2012), la deuda técnica intencionada puede ser fructífera si el coste de pago se mantiene visible y bajo control.

Por lo tanto, la deuda técnica es el resultado de diseñar, codificar e implementar un producto software en mal estado, lo cual abarca pérdidas monetarias. Pero no siempre es sinónimo de problema, ya que, vista desde otro enfoque, puede traer ventajas si es considerada como una estrategia para el mercado, tomando el debido control de la misma; de lo contrario tiende a acumularse, causando que el software no pueda ser escalable, elevando las tareas de mantenimiento, y en el peor de los casos el software debe ser refactorizado completamente.

Dentro de las posibles causas en las que se incurre el concepto de DT, las más comunes son:

- **Falta de conocimiento del concepto de la deuda técnica.-** Esto provoca que no se tomen en cuenta medidas necesarias al desarrollar un producto software, ya que induce la incapacidad del equipo de trabajo de desarrollar aplicaciones de alta calidad, además en el plan de desarrollo del producto no se suele toman en consideración un monto en el presupuesto para la deuda técnica para evitar pérdidas monetarias a futuro. Es por ello que debe considerarse un monto en el desarrollo de sus aplicaciones para disminuir la DT y ajustar los parámetros en sus estimaciones para determinar cuánto de esa deuda se puede reducir dentro del presupuesto disponible (Curtis et al., 2012a).

- **Ausencia del uso de buenas prácticas.-** Esto ocurre cuando no se toman en cuenta estándares necesarios en el desarrollo y mantenimiento de un software, generando pérdidas de calidad. Para desarrollar un producto de calidad se puede aplicar un modelo como ISO / IEC 25010, ISO /IEC 9126 o SQALE, los cuales se componen de características de calidad que se relacionan con requisitos establecidos para evitar que existan errores que afecten de forma negativa la entrega final el producto.

- **Priorización a ciertas funcionalidades del sistema.-** Cuando se concentran en la solución de ciertas funcionalidades que se consideran más importantes, dándole poca importancia a otras.

- **Mala estimación del cronograma de trabajo.-** Al planificar de manera incorrecta el cronograma de trabajo para el desarrollo de un producto, provoca un desfase en los tiempos de desarrollo, es decir, el equipo de desarrollo puede tomar atajos (induciendo a violaciones de buenas prácticas de diseño y codificación) y entregar un producto incompleto.

1.1.1 Clasificación de la deuda técnica.

Otras causas de acumulación de deuda técnica según el contexto de cada negocio expuesta por (Tom, Aurum, & Vidgen, 2013), son: deuda estratégica, deuda táctica y deuda incremental.

- **Deuda estratégica:** La deuda a largo plazo, por lo general sucede de forma proactiva, por razones estratégicas, como por ejemplo cuando se quiere ser el primero en el mercado optan por estrategias con fines de aprovechar el tiempo, obteniendo un pago a largo plazo.

- **Deuda táctica:** Esta deuda se toma de forma reactiva para corto plazo, como la toma de decisiones sub-optimas, por ejemplo cuando no se tiene tiempo de implementar una funcionalidad de manera correcta, y se la deja incompleta para corregirla en otro avance del proyecto.

- **Deuda Incremental:** Se describe como cientos o miles de pequeños atajos que se toman al desarrollar el software, por ejemplo nombres de las variables genéricas, comentarios dispersos, no seguir las convenciones de codificación, y así sucesivamente. Esta deuda se acumula fácilmente, y es difícil de rastrear y administrar.

Por ello es importante conocer los efectos de abordar alguna de estas tres deudas, sobre todo la que podría dar lugar a situaciones peligrosas, como la deuda incremental. Las dimensiones que reflejan problemas al incurrir a la DT son: la decadencia del código y diseño, así como una mala documentación y falta de pruebas. De tal modo que existen varios tipos de deuda, como: deuda de requisitos, deuda de código, deuda de diseño, deuda de medio ambiente, deuda de documentación y deuda de pruebas, las cuales se describen a continuación.

1.1.2 Tipos de deuda técnica.

La deuda técnica según las etapas del ciclo de vida de software se distribuye en los siguientes tipos:

- **Deuda de requisitos.-**Se refiere a la distancia entre la especificación óptima de requisitos y la implementación actual del sistema bajo limitaciones(Ernst, 2012), por ello deben ser claros, ya que, son la validación definitiva del éxito del proyecto y la manifestación de la parte interesada para el sistema (Curtis, Sappidi, & Szynkarski, 2012b).

- **Deuda de código**.- Se refleja en la mala escritura, o mal olor de código, el cual se puede eliminar a través de refactorización (Curtis et al., 2012a), como por ejemplo duplicaciones o exceso de código complejo.

- **Deuda de diseño y arquitectura.-** La cual se produce por decisiones sub-óptimas de arquitectura, que comprometen aspectos de calidad, especialmente cuando no se toma en cuenta la evolución del programa o la capacidad de mantenimiento.

- **Deuda de medio ambiente**.- Se manifiesta en el entorno de la aplicación, en el que incluye procesos relacionados con el desarrollo, así como del hardware e infraestructura, por ejemplo cuando se implementa el sistema en un equipo que no contenga las características necesarias para el perfecto funcionamiento del producto.

- **Deuda de la documentación**.-Se genera por la información desactualizada, incompleta o excesiva, por ejemplo si se realiza un cambio en una parte del sistema y no se documenta los cambios, esto genera dificultad en el desarrollo o en futuros mantenimientos del software.

- **Deuda de pruebas**.- Se ocasionan por los atajos que se toman en las pruebas o por falta de scripts de prueba.

Los defectos de software comúnmente no son visibles a los usuarios, sino sobre el código interno, y la calidad de la arquitectura, y suele presentarse cuando se da inicio a la evolución del software (Kruchten, 2012). Es por esta razón que el presente trabajo se centra particularmente en la **identificación de deuda de código y diseño**, ya que en estas etapas del ciclo de desarrollo se presentan comúnmente los problemas del software, además los fallos del software se pueden evidenciar a través de un análisis estático.

1.1.2.1 Deuda Técnica de Código.

Este tipo de deuda técnica se manifiesta en el código mal escrito que viola reglas de codificación(Li, Avgeriou, & Liang, 2015), como por ejemplo, código mal escrito por no alinearse a las buenas prácticas de programación, complejidad de métodos, duplicidad de código, falta de documentación del código, la ausencia de aplicación de patrones, el bajo nivel de testing del mismo, y la poca experiencia del desarrollador. (Villar & Matalonga, n.d.).

Estas anomalías de código permiten identificar en qué parte del desarrollo se ha diferido el esfuerzo, en el cual se acumula deuda técnica; así como este tipo de violaciones de las prácticas de programación, tales como: integración de módulos en etapas tardías, evitar copiar y pegar, escribir funciones o procedimientos demasiado largos y evitar el uso de variables globales. Las cuales deben ser corregidas a tiempo, debido a que contribuyen a altos costos de mantenimiento, tales como el esfuerzo excesivo para implementar los cambios.

1.1.2.2 Deuda Técnica de Diseño.

La deuda de diseño se refiere a las violaciones de arquitectura o fallas en el código fuente que afectan el diseño del software. (Seaman & Guo, 2011) manifiestan que, la deuda técnica arquitectónica es un tipo de riesgo importante en un proyecto de software a largo plazo. Se incurre a este término debido a soluciones sub-óptimas que se toman durante el diseño de la aplicación, los cuales comprometen a los atributos de calidad del software, especialmente el mantenimiento y capacidad de evolución (Li, Liang, & Avgeriou, 2014).

1.1.3 Causas de la acumulación de la deuda técnica Arquitectónica.

Las estructuras arquitectónicas de los sistemas a gran escala presentan defectos de diseño que se propaga y éstas estructuras erróneas son conocidas como las raíces de la deuda técnica arquitectónica, las cuales se generan por varias causas que producen su acumulación (Martini, Bosch, & Chaudron, 2014), entre las que constan:

- Factores de negocio
    - Las personalizaciones del producto crean cambios en la arquitectura, en la cual se acumulan automáticamente deuda técnica arquitectónica.
    - La presión del tiempo se da cuando hay acercamiento de fechas límites y fuertes multas por las entregas atrasadas.
    - Priorización de características con plazo de tiempo.
- Factores Técnicos
    - Mala documentación de arquitectura, es decir, la falta de especificación y/o énfasis en los requisitos arquitectónicos críticos.
    - La reutilización de sistemas heredados, terceros o de código abierto.
    - Equipos de desarrollo trabajando en paralelo, quienes automáticamente acumulan algunas diferencias en su diseño y arquitectura.
    - La refactorización incompleta no solo deja una parte de deuda arquitectónica, sino que creará nuevas deudas técnicas arquitectónicas.
    - La evolución de tecnologías, que hacen que se reemplacen junto con el software específico que necesita para funcionar en él.
    - El factor humano, debido a que, la acumulación de deuda puede también estar relacionado con decisiones sub-óptimas, debido a incapacidad, descuido o ignorancia.
    - En otro caso se manifiesta la deuda de diseño, cuando no se corrigen a tiempo las diferencias tecnológicas, haciendo necesaria una refactorización completa, debido a que el código se vuelve inmanejable (Heidenberg & Porres, 2010). También se indica que, la deuda técnica puede ser evaluada mediante la medición de la salubridad de la arquitectura de software(Nord, Ozkaya, Kruchten, & Gonzalez-Rojas, 2012), que es un indicador importante del costo futuro de trabajo(Guo, Spínola, & Seaman, 2016).

1.2 Gestión de la Deuda Técnica.

Al tener conciencia de la existencia de deuda técnica en el software, se pueden tomar dos decisiones: dejar el producto tal y como está y asumir ese pago de intereses o plantearse reducir la deuda y gestionarla. Si se escoge la segunda opción, se debe conocer cuál es el mecanismo de seguimiento y gestión adecuado. Según (Guo et al., 2016), el proceso de gestión de la deuda técnica consiste de tres actividades generales: identificación, medición y monitoreo, las mismas que pueden ayudar a los administradores de software a tomar decisiones para sus proyectos.

1.2.1 Actividades de la gestión de la deuda técnica.

Los problemas de la deuda técnica en un proyecto de software no se toman en consideración hasta que se salen de control, los resultados pueden ser graves y pueden conducir al fracaso del proyecto, por lo tanto es necesario considerar las actividades que se pueden ejecutar en la gestión de la deuda técnica. Por ello en esta sección se presentan los conceptos básicos de las actividades que se deben tomar en cuenta en la gestión de la deuda técnica. Las actividades de la gestión de la deuda técnica que se manejan de manera secuencial son: identificación, medición y monitoreo. (Ver Figura 1)

- **Identificación.-** Detecta causas de la deuda técnica en un sistema de software a través de técnicas específicas, tales como el análisis de código estático.
- **Medición.-** Cuantifica el costo y beneficio de la deuda técnica conocida en un sistema de software, a través de técnicas de estimación.
- **Monitoreo.-**Observa los cambios de costes y beneficios no resueltos que se producen durante el tiempo.

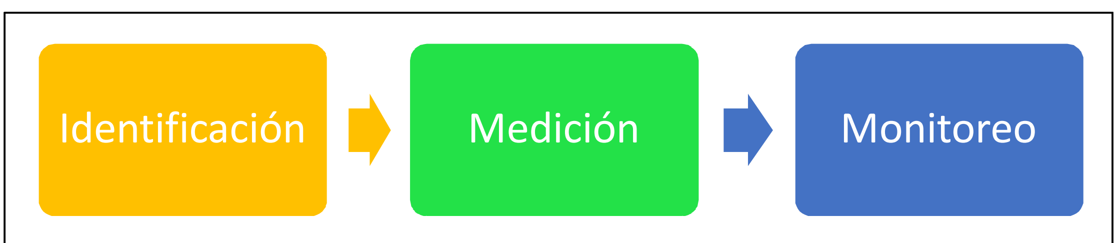


Figura 1.Actividades de la gestión de la deuda técnica
Fuente: (Guo et al., 2016)
Elaboración: El autor

Como se observa en la Figura 1, existen varias actividades que se puede ejecutar para mantener un sistema de software saludable, tanto en ámbitos de corrección (priorización), como de prevención; en el presente trabajo se enfocará en los procesos de identificación y medición de la deuda técnica, debido a que el monitoreo de un software se lo maneja mediante iteraciones, los cuales requieren de tiempo y una metodología para su control iterativo.

Las actividades de identificación y medición se pueden realizar ya sea en el desarrollo del producto como en la implementación del mismo. La identificación se la realiza mediante la aplicación de un análisis estático al código fuente, en el que se obtienen los errores de código y/o diseño del software. La medición se la realiza en base a los resultados obtenidos de la identificación. Y el monitoreo se lo realiza al producto final una vez que se hayan corregido los errores encontrados en el software.

1.2.1.1 Identificación de la deuda técnica.

Es un proceso de evaluación de bloques defectuosos en el código fuente de un software, por lo tanto en esta actividad se reconoce la existencia de deuda técnica mediante un análisis de calidad de software (Guo et al., 2016) en el que se identifica el código sucio y el incorrecto diseño del software. Esto se lo puede realizar mediante herramientas de análisis de calidad de software.

El objetivo de identificar la deuda técnica es poder tomar decisiones; es decir, cuando se identifiquen las instancias de deuda técnica y el potencial de daño que tengan, se podrá decidir qué elementos técnicos de deuda deben ser pagados o corregidos (Seaman et al., 2012). Es por ello que este paso es de gran importancia, debido a que es una de las primeras actividades que se manejan en la gestión de la deuda técnica, en donde se conoce de manera directa si un software es defectuoso.

Para identificar la deuda de código y la deuda de diseño, primero se definen métricas a evaluar, para posteriormente realizar el análisis de DT, a través de un análisis estático al código fuente o ejecutable de la aplicación tal como lo requiera la herramienta de análisis de calidad de software.

En caso de ejecutar un análisis de deuda técnica a nivel de código y diseño en la misma aplicación, se lo puede realizar siempre y cuando se clasifique los resultados obtenidos; esta

clasificación se la puede efectuar en una hoja de cálculo, en la que se etiquete a cada violación detectada con el tipo de deuda a la que pertenece.

1.2.1.2 Medición de la deuda técnica.

La medición de la deuda técnica es un proceso de cuantificar o calcular el costo de solución de las fallas del código o diseño del software (Nord et al., 2012). Posteriormente al proceso de identificación de los defectos del software, y en base a los resultados obtenidos de las fallas del software, como segundo paso se calcula el esfuerzo necesario para corregir dichas falencias del código.

Otro enfoque de medición de la deuda es mediante la visualización de la edad del código fuente, ya que de esta manera se puede identificar si es necesario una refactorización del sistema (Thiele, 2014).

La cuantificación de la deuda técnica se la puede realizar tanto a nivel de código o de diseño del software, en base a los resultados de las violaciones de clases y/o arquitectónicas que se encuentran en la fase de identificación, ya que el valor de DT se mide mediante la gravedad de los errores encontrados (estos dependen del número de violaciones encontradas por cada regla). Así mismo al igual que la identificación, los resultados se clasifican, de tal manera que se identifique el tipo de DT (código o diseño).

1.2.1.3 Monitoreo de la deuda técnica.

Una vez solucionados los problemas de deuda técnica detectados en el software, es necesario considerar el seguimiento al software corregido para verificar que se ha reducido la deuda, ya que por el contrario se verán comprometidos los atributos de calidad del software, tal como capacidad de modificación, es por ello que es necesario un seguimiento apropiado enfocándose a los atributos de calidad.

El monitoreo de la deuda utiliza de entrada el código fuente del software ya corregido con el objetivo de visualizar los cambios de costo y beneficio no resueltos que se producen a lo largo del tiempo. Esta práctica de monitoreo con enfoque en los atributos de calidad (Bellomo, Nord, & Ozkaya, n.d.), se centran en: periódicas revisiones de diseño, el uso de estándares y modelos de referencia, manejo de errores o de supervisión del rendimiento.

Se dice que la medición de la deuda técnica se hace generalmente con métodos de ingeniería de software, pero el monitoreo se lo realiza con el seguimiento de portafolios, y el reembolso con el análisis coste / beneficio (Ampatzoglou, Ampatzoglou, Chatzigeorgiou, & Avgeriou, 2015).

Se puede ver que estas prácticas de medición y monitoreo son de gran importancia, debido a que su aplicación garantiza que un sistema de software sea saludable, evitando de esta manera pérdidas monetarias a futuro.

Para una mejor obtención de resultados en el análisis de deuda técnica de código y diseño, se puede identificar mediante técnicas que ayudan en este proceso, como: análisis estático, code smells, anti patrones de diseño y suciedad acumulada y violaciones de modularidad (enfocados a grandes sistemas de software). (Seaman, 2013)(Zazworka et al., 2014).

1.3 Análisis estático.

El análisis estático es el proceso que permite analizar el software sin necesidad de ejecutarlo, para ello utiliza como entrada el código fuente, permitiendo encontrar problemas tales como: fallos en las propiedades de diseño, duplicidad y complejidad de código, entre otras. Con la información que se obtiene del análisis estático se puede conocer qué mejoras se deben efectuar tanto de código como de diseño del software, minimizando la deuda técnica encontrada y otorgando beneficios a corto y largo plazo (pensando en el mantenimiento del producto).

El número de problemas estructurales que se pueden identificar en una aplicación, se los puede medir a través del análisis estático del código fuente (Curtis et al., 2012b), para ello se sugiere la ponderación de los resultados(es decir, los problemas detectados), clasificándolos por categorías, como: severidad del daño, cantidad de violaciones, tipo de DT, para así conocer cuáles son los que necesitan ser corregidos.

Existen un sinnúmero de herramientas que realizan análisis estático de forma automatizada, las cuales realizan la búsqueda de violaciones de clases, arquitectura y buenas prácticas de programación. Algunas de estas herramientas además de evaluar y detectar fallas de código y de diseño, calculan el grado de esfuerzo necesario para corregir las fallas del software.

Estas herramientas de análisis estático, siguen un mismo proceso al momento de trabajar con una aplicación; realizando controles basados en el análisis léxico que procesan el código fuente; en el cual se lo analiza mediante un conjunto de reglas de programación. (Díaz & Bermejo, 2013). El proceso que siguen las herramientas al realizar un análisis estático al software es el siguiente:

1. Transformar el código para analizarlo en un modelo de programa, un conjunto de estructuras de datos que representan el código.
2. Analizar el modelo usando diferentes reglas y / o propiedades.
3. Visualizar los resultados proporcionados por la herramienta utilizada.

Características del análisis estático:

- El análisis estático se lo puede realizar de forma manual o automática (mediante una herramienta de análisis de calidad de código).
- No es necesario la compilación del software.
- Las herramientas para realizar un análisis estático contienen numerosas reglas de programación para poder encontrar más tipos de bugs[1].
- Se puede realizar este tipo de análisis durante el desarrollo del software.
- Se adapta a cualquier tipo de lenguaje de programación.
- Puede analizar errores en la estructura del software.

La Tabla1 presenta algunas ventajas y desventajas de la aplicación de un análisis estático sobre una aplicación de software.

Tabla1. Ventajas y desventajas del análisis estático.

Ventajas	Desventajas
Mejora la calidad del código fuente.	Permite conocer cómo mejorar el código, pero no sabemos si es funcional.
Existen herramientas para detectar errores de forma automática, sin la ejecución del código.	Hay problemas que son difíciles de encontrar de forma automática.
Se puede integrar con IDE'S	Las herramientas se centran en algunos lenguajes de programación en específico.
Disminuye los costos en los procesos de calidad de QA.	Pueden devolver falsos positivos
No es necesario la compilación del código.	No garantiza que funcione la aplicación.

[1] Un bug hace relación a un error de software.

Se aplica a cualquier tipo de software.	Es necesario complementar el análisis con para poder realizar mejoras en un ámbito mayor.

Elaboración: El Autor

1.4 Code Smells.

El concepto de Code Smells (también conocido como mal olor de código) fue introducido por primera vez en 1999 y describe a los sistemas orientados a objetos que no cumplan con los respectivos principios (por ejemplo, ocultación de información, la encapsulación, el uso de herencia)(Fowler, 1999).

El mal olor de código (code smells) presenta síntomas de posibles daños en el código o problemas de diseño, que se puede eliminar a través de refactorización[2]. (Fontana, Ferme, & Zanoni, 2015)

Se puede decir que un software suele presentar problemas minúsculos, pero que en realidad son más profundos, no por ello significa que dejen de funcionar, sin embargo puede ser que se causen daños a futuro.

Características de Code Smells:

- El tener código duplicado es un síntoma de code smells, así como el código muerto, es decir, que no tiene ninguna función.
- Se producen este tipo de deficiencias al crear variables con nombres parecidos.
- Los métodos o funciones largas son difíciles de entender, lo cual genera dificultades de mantenimiento.
- El code smells del software puede ser corregido mediante la refactorización.
- Un code smells no es necesariamente un bug, porque puede que el programa está funcionando normalmente.
- Se presencia code smells en el software cuando el código es difícil de entender.

La Tabla *2* expone algunas de las ventajas y desventajas al momento de utilizar análisis estático para identificar code smells sobre una aplicación de software.

[2] Una refactorización es una corrección que mejora la calidad del software.

Tabla 2. Ventajas y desventajas de Code Smells.

Ventajas	Desventajas
Es fácil para las personas sin experiencia para detectarlos, incluso si no saben lo suficiente para evaluar si hay un problema real o corregirlo.	Existen “olores” que no son fáciles de detectar.
Las herramientas de análisis de code smells pueden ayudar a incorporar actividades de refactorización.	A veces se pueden acelerar a realizar una refactorización, pero es necesario encontrar más de un problema para poder realizar una refactorización.
Indica deficiencias en el diseño del software.	La identificación de problemas de diseño puede ser difíciles de encontrar.
La detección de code smells evita problemas a futuro.	No siempre son errores de programación.

Elaboración: El Autor

1.5 Anti patrones de diseño y acumulación de suciedad.

Los patrones de diseño son soluciones a problemas de diseño general en un contexto particular y la clave del uso de los mismos es poder facilitar las tareas de mantenimiento y la posibilidad de tener un software escalable. Pero al no saber implementarlo correctamente se produce suciedad en los mismos, la suciedad de los patrones consiste en la acumulación de artefactos que no se relacionan a las clases principales de los patrones de diseño, o sea, estos artefactos no contribuyen a la función prevista de un patrón de diseño convirtiéndolos en anti patrones (Izurieta & Bieman, 2008).

Los patrones de diseño son muy utilizados, pero presentan problemas, debido a que no son inmunes a las fallas durante su evolución, ya que la acumulación de suciedad en los patrones, ofusca la estructura y el comportamiento original del patrón. (Griffith & Izurieta, n.d.) Además la utilización de anti patrones compromete a la capacidad de prueba, y que a medida que se acumula la suciedad, los casos de prueba se rompen (Seaman, 2013).

Por ello es necesario evitar el crecimiento de suciedad en los patrones de diseño, los cuales se forman por el incremento de código que no forma parte de ningún patrón, causando problemas en la etapa de pruebas del software; debido a que el objetivo del uso de patrones es que el software sea menos propenso a defectos.

Características de anti patrones de diseño:

- Se origina la suciedad en patrones de diseño cuando se crean clases que no tienen relación con ningún patrón.

- Son vulnerables a fallas durante el desarrollo de software.
- La suciedad de patrones provocan problemas de testing.
- Se consideran fallas de diseño del software.
- Al utilizar un anti patrón en el software es necesario su documentación, en el que debe contener la solución que ofrece, lo negativo de usarlas y como recuperarse de los problemas que genera.
- Se considera también un anti patrón a la aplicación correcta de un patrón de diseño en el contexto equivocado

La Tabla *3* menciona las ventajas y desventajas de identificar el uso de anti-patrones de diseño de una aplicación de software.

Tabla 3. Ventajas y desventajas de la detección de suciedad en patrones de diseño.

Ventajas	Desventajas
Dicho análisis ayuda a mejorar la estructura del software.	Mientras el software se desarrolla, la acumulación de suciedad en patrones pueden impedir la detección de fallas
Detecta dependencias ocasionadas por la mala utilización de patrones.	Son trampas consideradas malas soluciones de problemas de diseño.
La documentación de anti patrones ayuda a los diseñadores de software a no tomar malos caminos.	Presentan más problemas que soluciones.
El estudio de anti patrones permite conocer los errores más comunes al desarrollar un programa.	Se basan principalmente en la solución a un problema y terminan siendo el problema.

Elaboración: El Autor

1.6 Violaciones Modulares.

Una violación se considera modular cuando existen dos componentes que pertenecen a diferentes módulos pero siempre cambian juntos (Wong, Cai, Kim, & Dalton, 2011). Es por ello que es necesario descubrir las relaciones ocultas entre los archivos (Kazman, Cai, Mo, Feng, & Xiao, 2015) mediante un análisis en el diseño del software.

Las violaciones modulares son consideradas problemas de diseño y afectan en mayor cantidad a los sistemas de software grandes, debido a que los módulos están diseñados para evolucionar independientemente, pero por el contrario lo hacen iguales, y por lo tanto estas fallas se deben a cambios que se dan al diseño original (Zazworka et al., 2014).

Para identificar la deuda técnica de diseño, las herramientas de análisis de calidad efectúan la detección de violaciones modulares y el análisis de clases (Fern & Garbajosa, 2015). La detección de violaciones modulares es de gran utilidad, para detectar errores en proyectos de gran escala, y así se puedan corregir los defectos de dependencia.

Características de violaciones modulares:

- Se producen cuando los componentes son dependientes o están relacionados con otros que cumplen otras funciones.
- Estas dependencias se dan cuando existe clonaciones de código.
- Los desarrolladores inexpertos pueden provocar este tipo de problemas al olvidarse de eliminar el código provisional.
- Se generan cuando la acoplar los módulos de manera incorrecta.
- En caso de que un módulo necesite de otro, se pueden relacionar mediante una interfaz de comunicación bien definida.
- La corrección los problemas modulares pueden ser muy complejos, dependiendo del tamaño del software.

La Tabla 4 indica las ventajas y desventajas que se tienen realizar un análisis estático para identificar violaciones modulares del software.

Tabla 4. Ventajas y desventajas de la detección de violaciones modulares.

Ventajas	Desventajas
Mejora la calidad del código fuente.	Al examinar un sistema grande, es posible que el análisis tenga mayor dificultad.
Pueden encontrarse falsos positivos.	Se pueden encontrar falsos negativos, tas un análisis.
Mejoran las tareas de mantenimiento.	Algunas estructuras no se adaptan a las herramientas de análisis.
Permiten la fácil evolución del software.	Se elevan los costos de mantenimiento.

Elaboración: El Autor

1.7 Herramientas para la identificación de deuda técnica.

Existen varias herramientas que facilitan las tareas de identificación de la deuda técnica, las cuales permiten obtener información sobre dónde se encuentran fallas y cuál es el potencial de las mismas, para poder tomar decisiones de corrección. En la Tabla 5 se listan una serie

de herramientas que permiten realizar análisis de código asociadas a técnicas de análisis estático para identificar la DT (deuda técnica).

Tabla 5. Herramientas para la gestión de la deuda técnica.

Nombre	Técnica de análisis	Tipo de deuda técnica	Comerciales	No comerciales	Lenguajes de programación soportados
CAST Application Intelligence Platform (AIP)	Análisis estático	DT de código, DT arquitectónica	X		Java, Oracle Forms, .NET, C++
SonarQube	Análisis estático	DT de código		X	Javascript, PHP, Cobol, PL, C#,SQL
FindBugs	Análisis estático	DT de código		X	Java
Structure 101	Análisis estático	DT arquitectónica	X		C, C++,C # , Python, Pascal
Understand	Violaciones Modulares	DT de código	X		C/C++, C#, Python
Sonargraph	Análisis estático	DT de código DT arquitectónica	X		Java
Source meter	Análisis estático	DT de código	X		C, C++,C # Java, Python.
NDepend	Análisis estático	DT arquitectónica	X		C#
Lattix	Análisis estático	DT arquitectónica	X		C ++, .NET, Java
SourceMonitor	Análisis estático	DT de código		X	C, C++,C # #, VB.NET, Java, Delphi, Visual Basic
SQUORE	Análisis estático	DT de código	X		Ada, C++,C # Java, PL, SQL, Python
Kiuwan	Análisis estático	DT de código	X		Java, JSP, Javascript, PHP, C, C++, ABAP, COBOL, JCL, C#, PL/SQL
Visual Paradigm		DT arquitectónica	X		UML, SysML, ERD, BPMN, DFD, ArchiMate
PMD	Análisis estático	DT de código		X	Java
MIA-Quality	Análisis estático	DT de código	X		Java

Elaboración: El Autor

A continuación se realiza una descripción de las herramientas expuestas en la Tabla 5, en el que se indica los lenguajes que soporta y la técnica de análisis que utiliza.

- **CAST’s Application Intelligence Platform (AIP)** [3]

Es una plataforma que es utilizada para la identificación de violaciones de código fuente y la categorización de las violaciones por atributos de calidad. CAST AIP además de cuantificar la DT proporciona al equipo de desarrollo herramientas necesarias para gestionarla y reducirla.

(Curtis et al., 2012a) y (Alzaghoul, 2014) en sus trabajos utilizan la herramienta para detectar altos componentes con dependencia mediante el análisis del código fuente. La Figura 2 presenta la interfaz de usuario de la herramienta CAST.

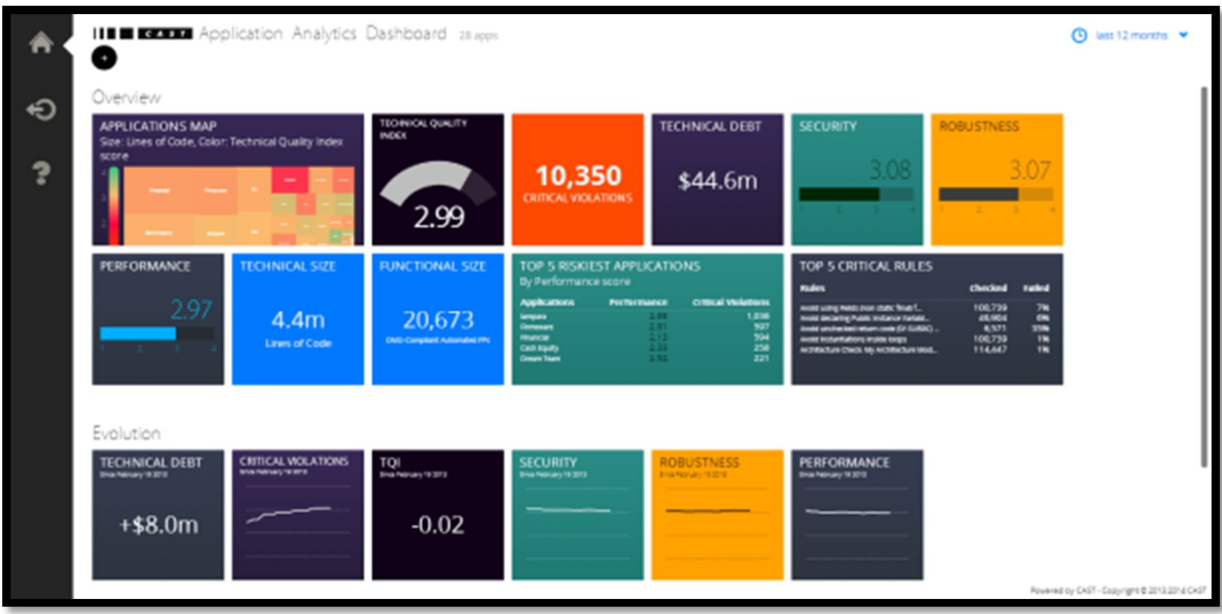


Figura 2.Interfaz de usuario de la herramienta CAST AIP.

Fuente: (“Cast Software,” n.d.)

Características de la herramienta:

- Los lenguajes que evalúa la herramienta son los siguientes: SQL-PSM, VB, T-SQL, PL-SQL, PowerBuilder, JSP, Java, Oracle Forms, .NET, C++, COBOL, ASP, y ABAP.
- Analiza métricas como: complejidad, dependencias, código muerto, tamaño de software, uso e comentarios.
- Realiza análisis de deuda técnica.
- Realiza un control de seguridad en las aplicaciones de software.
- Técnica: Análisis estático.

[3]http://www.castsoftware.com/

- **SonarQube**

SonarQube es una plataforma para la gestión de la calidad de código(Li et al., 2015). La herramienta identifica la deuda como grupos arquitectónicamente relacionados, utilizando indicadores como: líneas de código, la complejidad, y líneas duplicadas (Kazman et al., 2015). Eisenberg en su investigación utiliza Sonar para calcular las interdependencias de paquetes con el fin de identificar los posibles paquetes fuertemente acoplados (Eisenberg, 2012).

(Mamun & Berger, 2014) incluye el uso de SonarQube, una herramienta de análisis de la deuda técnica. SonarQube calcula la deuda técnica fundamentada en la evaluación de la calidad del software, basado en la metodología del ciclo de vida Expectativas (SQALE). La deuda técnica puede ser detectada y evaluada usando análisis estático, tal como la herramienta Sonar(Kruchten, 2012). En la Figura 3 se muestra la interfaz de usuario de la herramienta SonarQube.

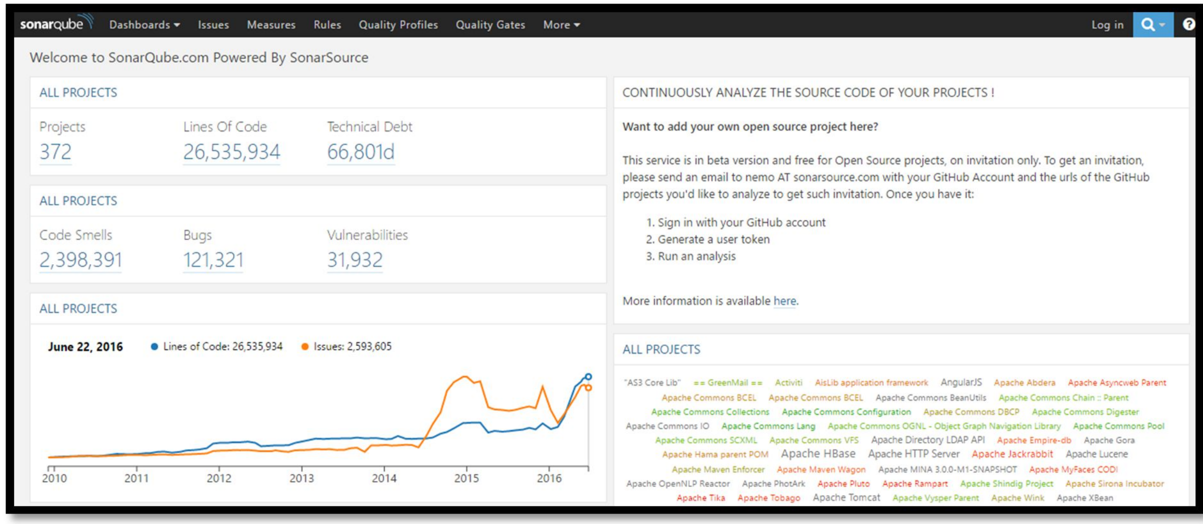


Figura 3.Interfaz de usuario de SonarQube[4].

Fuente: ("SonarQube," n.d.)

Características de la herramienta:

- Se pueden hacer análisis sobre: código duplicado, estándares de codificación , tests, cobertura de pruebas, complejidad ciclomática, bugs potenciales, comentarios, diseño y arquitectura.
- Realiza análisis de deuda técnica.

---

[4]http://www.sonarqube.org/

- Permite el acoplamiento de otras herramientas como Findbugs, Sonargraph, PMD mediante plugins.
- Analiza métricas como: complejidad ciclomática, dependencias, acoplamiento, tamaño de software, uso e comentarios, duplicaciones.
- Técnica: Análisis estático.

- **FindBugs**[5]

Es un software de código abierto, que mediante un análisis estático busca errores en programas desarrollados en lenguaje Java.

(Vetro, Morisio, Torchiano, & Torino, 2008) en un estudio analiza 301 proyectos desarrollados en Java, mediante la herramienta FindBugs, recolectando problemas señalados en el código fuente. Posteriormente, se comprobó la precisión de los problemas con cambios en la información.. En la Figura 4 se puede observar un ejemplo del funcionamiento de la herramienta FindBugs.

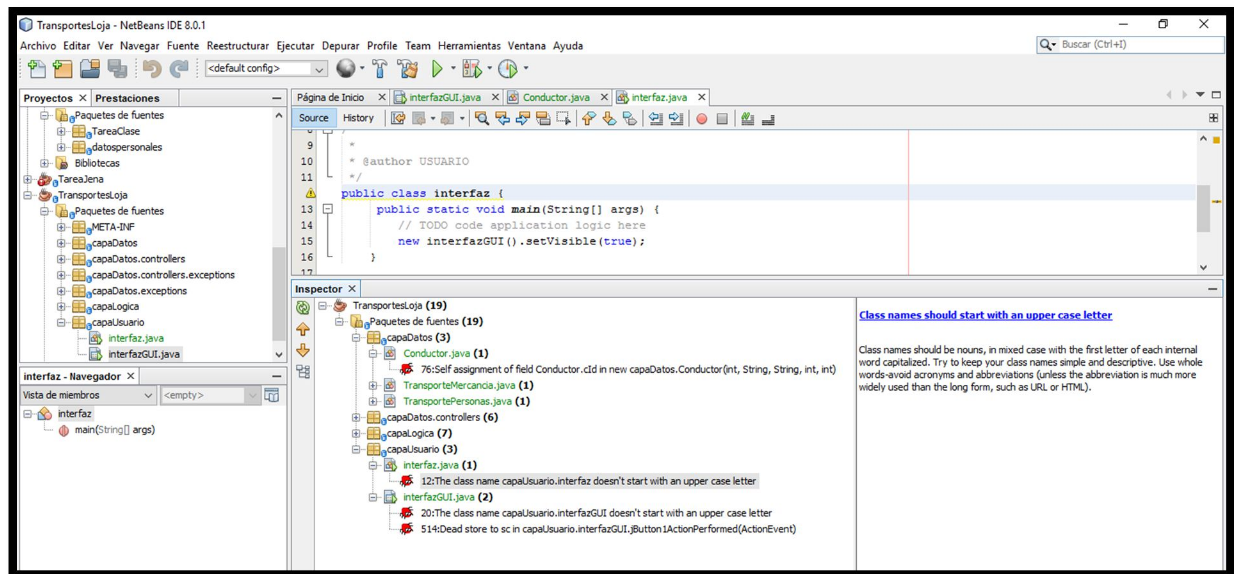


Figura 4.Ejemplo de análisis con plugin FindBugs en NetBeans.

Fuente: (“FindBugs,” n.d.)

Características de la herramienta:

- Esta herramienta clasifica los errores en tres categorías: correcto pero probable bug, malas prácticas y código dudoso.

[5]http://findbugs.sourceforge.net/

- Analiza métricas como: seguridad, duplicaciones, acoplamiento, tamaño de software, uso e comentarios, complejidad.
- No realiza análisis de deuda técnica.
- Se adapta a Netbeans y a SonarQube.
- Técnica: Análisis estático.

- **Structure 101**[6]

Structure 101 tiene una serie de herramientas a su disposición que ayudan a mejorar la modularidad del código base. Los arquitectos de software se pueden apoyar de la herramienta para mantener normas de arquitectura, diagramas que organizan el código en una jerarquía modular. En la Figura 5 se visualiza la interface de la herramienta Structure 101.

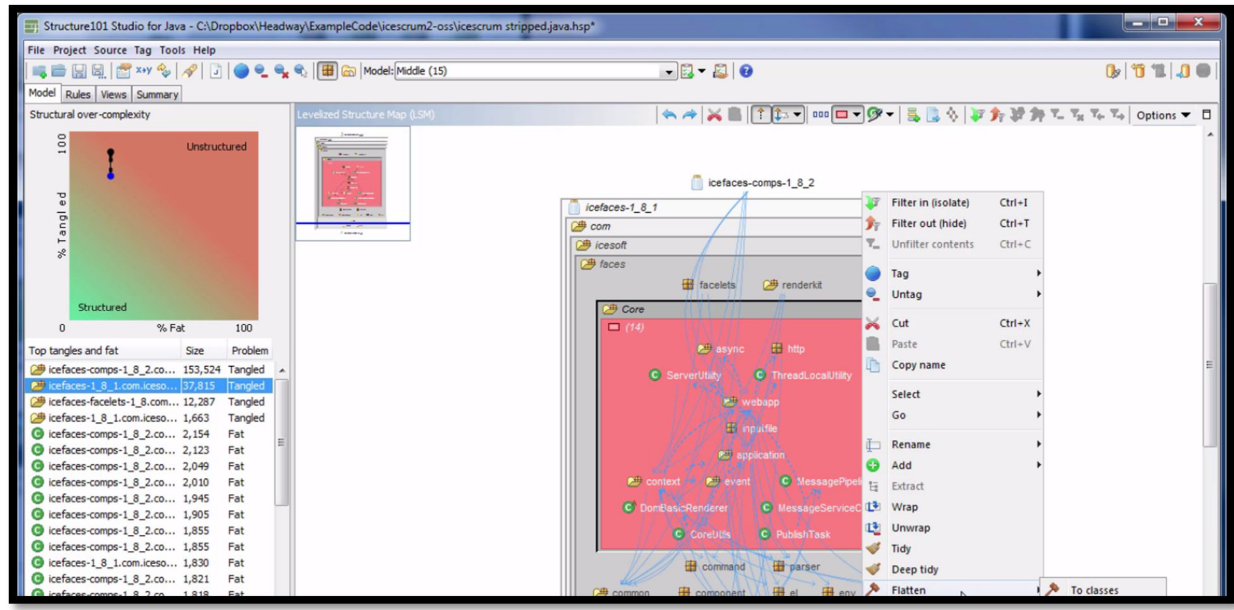


Figura 5. Interfaz de usuario de Structure 101.

Fuente: (“Structure101,” n.d.)

Características de la herramienta:

- Trabaja con los siguientes lenguajes: Ada, Cobol, ANSI C, K & R C, ANSI C ++, C #, FORTRAN, Java, Jovial, Pascal, PL / M, Python, VHDL, Objective C, Objetivo C ++, HTML, PHP, JavaScript, XML.
- Se adapta al IDE Ecplise mediante un plugin.

[6]http://structure101.com/

- Utiliza estructuras existentes para el análisis del software, tal como paquetes de proyectos Maven
- No realiza análisis de deuda técnica.
- Analiza métricas como: complejidad, violaciones de arquitectura, dependencias.
- Técnica: Violaciones Modulares.

- **Understand**[7]

Understand permite el análisis estático de código centrado en la comprensión del código fuente, métricas, y los estándares de pruebas. También permite ver rápidamente toda la información sobre las funciones, clases, variables, etc. y la forma en que se utilizan, cuáles han sido modificados y cómo interactúan.

Además permite visualizar todas las dependencias en su código, cómo se conectan y poder exponerlos de forma gráfica. En la Figura 6 se puede ver un ejemplo de análisis de software con la herramienta Understand.

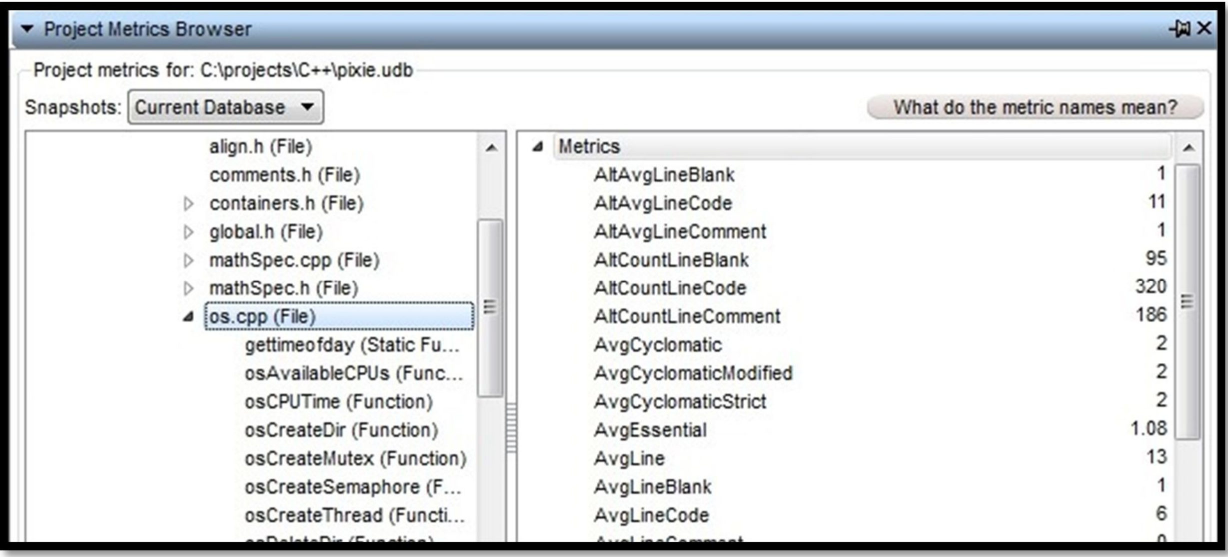


Figura 6.Ejemplo de análisis de métricas con Understand.

Fuente: (“Understand,” n.d.)

Características de la herramienta:

- Buscar problemas estructurales y violaciones importantes de las mejores prácticas de codificación.

[7]https://scitools.com/

- Analiza métricas como: complejidad, tamaño de software, número de módulos, número de métodos, acoplamiento de clases.
- Es posible escribir sus propias métricas personalizadas.
- No realiza análisis de deuda técnica.
- Técnica: Análisis estático.

- **SonarGraph**[8]

La herramienta SonarQube es escrita en Java, la cual permite visualizar gráficamente las dependencias existentes en un programa. Asimismo se pueden transferir los datos de análisis de Sonargraph a sonar automáticamente y mostrar estos datos en el cuadro de mandos Sonar. En la Figura 7 se presenta un ejemplo de visualización de dependencias con la herramienta SonarGraph.

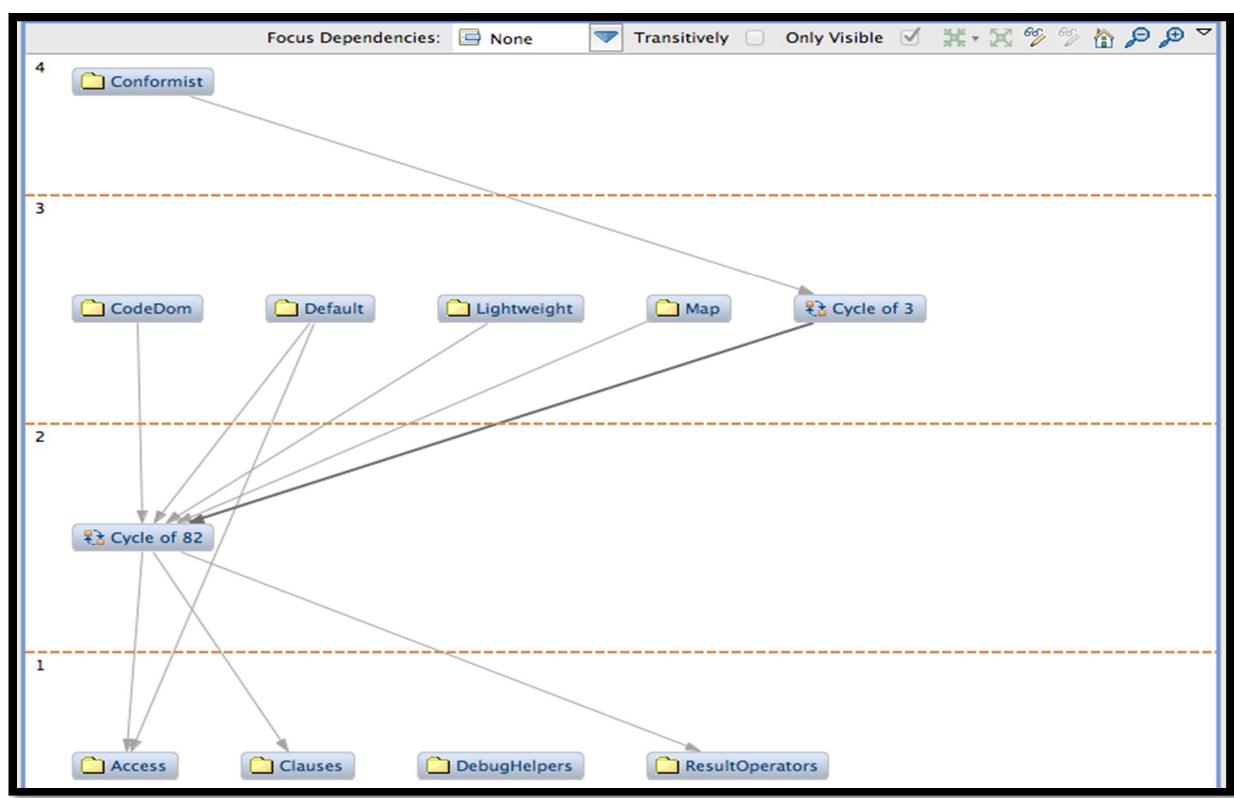


Figura 7.Ejemplo de visualización de dependencias con SonarGraph.

Fuente: ("SonarGraph," n.d.)

Características de la herramienta:

- Detecta violaciones de arquitectura.

---

[8]http://docs.sonarqube.org/display/SONARQUBE45/Sonargraph+Plugin

- Está disponible una versión gratuita del plugin en SonarQube para realizar pruebas de análisis.
- Tiene aplicación de escritorio, con versión gratuita hasta 14 días.
- Analiza métricas como: líneas de comentarios, componentes con dependencia, complejidad ciclomatica, componentes ignorados, duplicaciones.
- No realiza análisis de deuda técnica.
- Técnica: Análisis estático.

- **SourceMeter**

Esta herramienta proporciona un análisis estático a partir del código fuente. Permite encontrar los puntos débiles de un sistema en desarrollo solo con el código. También se integra a herramientas de análisis de código como FindBugs y presentar los resultados de manera unificada. En la Figura 8 se puede ver como se realiza un análisis de calidad de software con la herramienta Sourcemeter.

```
426             for (int i = 0; i < revChars.length; ++i) {
427                     switch (revChars[i]) {
428                     case '^':
429                             if (rev == null) {
430                                     if (name == null)
        SourceMeter (from PMD): Avoid using if statements without curly braces
        Open | Debt: 30min
431                                             if (done == 0)
432                                                     name = new String(revChars, done, i);
433                                             else {
434                                                     done = i + 1;
435                                                     break;
436                                             }
437                                     rev = parseSimple(rw, name);
438                                     name = null;
439                                     if (rev == null)
440                                             return null;
441                             }
```

Figura 8. Ejemplo de análisis de código con SourceMeter[9].

Fuente: (“SourceMeter,” n.d.)

Características de la herramienta:

- Realiza un análisis estático preciso y profundo.
- Trabaja con los siguientes lenguajes: C/C++, Java, C#, Python.

[9]https://www.sourcemeter.com/

- Tiene una versión gratuita pero con funcionalidad limitada.
- Analiza métricas como: duplicaciones, complejidad, cohesión, herencias.
- No realiza análisis de deuda técnica.
- Técnica: Análisis estático.

- **NDepend**[10]

Es un plugin de SonarQube escrito en Java, el cual permite visualizar gráficamente las dependencias existentes en un programa. También permite manejar grandes proyectos y obtener un resumen de los mismos. Por lo tanto, sin depender del hecho de que tan grande es el conjunto, cuántos tipos hay en su montaje, la cantidad de líneas de código hay en su proyecto; se puede determinar fácilmente la calidad del código y cuáles son las secciones que el código debe ser contempladas y refactorizadas. La Figura 9 indica un ejemplo de resultados que presenta la herramienta Ndepend.

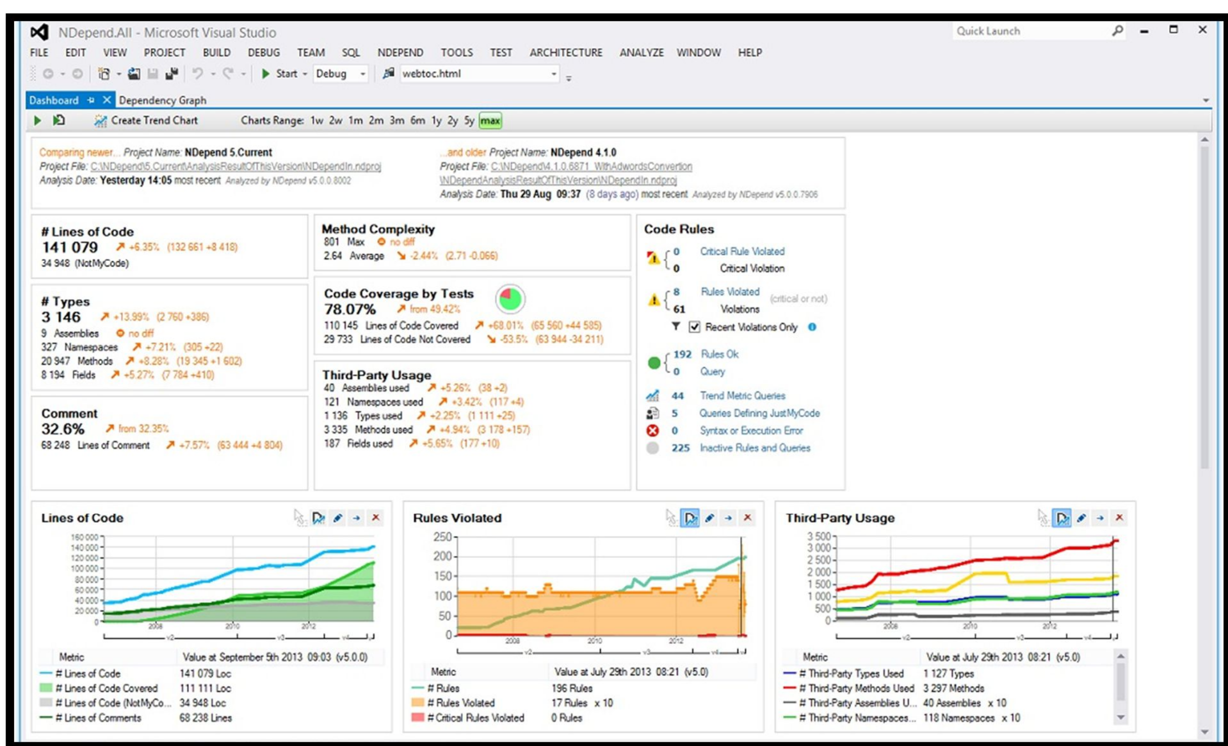


Figura 9. Interfaz de usuario de Ndepend.

Fuente: (“NDepend,” n.d.)

[10]http://www.ndepend.com/

Características de la herramienta:

- Tiene más de 150 reglas del código por defecto para comprobar con las mejores prácticas, como: líneas de código, comentarios, acoplamiento, testing, complejidad ciclomatica.
- El lenguaje que soporta Ndpend, es .NET.
- No realiza análisis de deuda técnica.
- Detecta ciclos de dependencia.
- Revisa la estructura del sotware e indica problemas arquitectónicos existentes.
- Técnica: Análisis estático.

- **Lattix**[11]

Esta herramienta ofrece un software de mayor calidad, acelera los plazos de desarrollo y reduce los costos de trabajo, además permite visualizar gráficamente las dependencias existentes en un programa y se integra perfectamente en el ciclo de vida de desarrollo. Lattix analiza dependencias entre los artefactos de un proyecto de software y analiza su arquitectura en detalle. La Figura 10 muestra un ejemplo de análisis con Lattix, en el que se presentan métricas del tamaño del software.

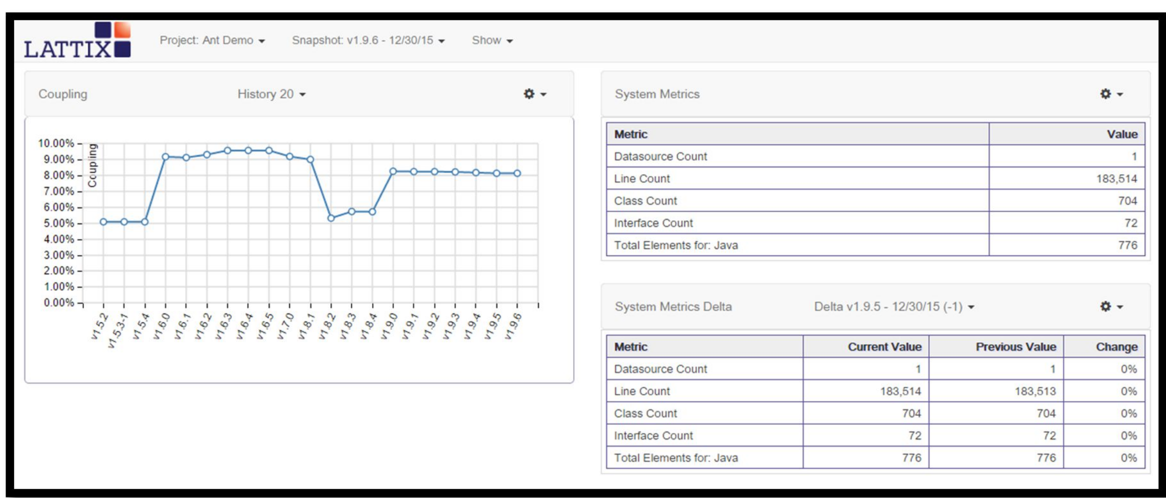


Figura 10. Ejemplo de reporte de análisis de calidad de software de Lattix.

Fuente: (“Lattix,” n.d.)

Características de la herramienta:

- Mejora problemas de modularidad.
- Identifica problemas de dependencias no deseadas.

[11]http://lattix.com/

- Analiza métricas como: complejidad ciclomatica, dependencias, violaciones de diseño.
- Analiza el impacto luego de una factorización.
- Soporta los lenguajes: C, C++, Java.
- Técnica: Análisis estático.

- **SourceMonitor**[12]

Es una herramienta que mide la cantidad y la calidad de código fuente, además proporciona la persistencia de los resultados del análisis y la comparación histórica de éstos los resultados guardados. También SourceMonitor presenta una visualización detallada de las métricas para cualquier archivo o punto de control del proyecto. En la Figura 11 se presenta un ejemplo de resultados de la herramienta, en el que indica mediante una gráfica el aumento de líneas de código en las diferentes versiones de un software.

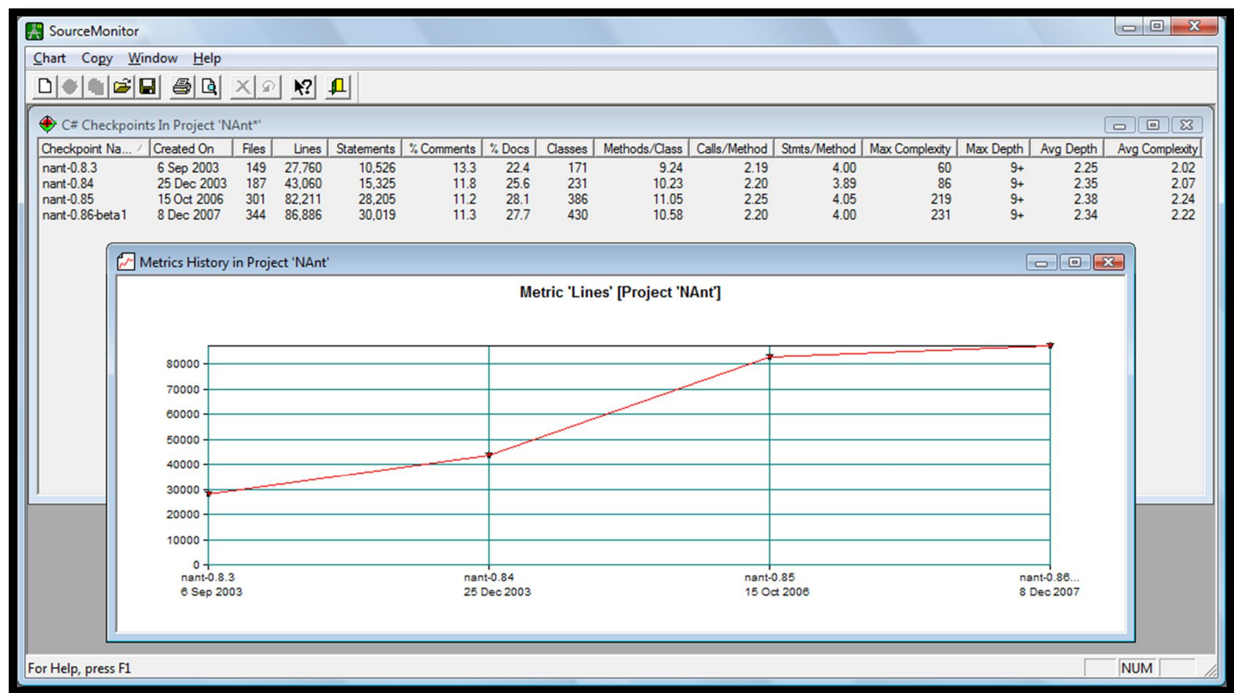


Figura 11. Ejemplo de funcionamiento de herramienta SourceMonitor.

Fuente: (“SourceMonitor,” n.d.)

Características de la herramienta:

- Presenta resultados del análisis de métricas en formatos XML o CSV.

[12] http://www.campwoodsw.com/sourcemonitor.html

- Muestra e imprime las métricas en tablas o gráficos.
- Cuenta con una interfaz gráfica para su control.
- No realiza análisis de deuda técnica.
- Analiza métricas como: complejidad ciclomatica, comentarios, tamaño de software.
- Técnica: Análisis estático.

- **SQUORE**[13]

Es una herramienta que permite la gestión de calidad de software, incluyendo la gestión de la deuda técnica. Posee indicadores vienen con modelos listos para el uso de calidad que se pueden personalizar fácilmente para adaptarse al contexto de su proyecto. En cuestión de minutos, la solución permite desplegar un nivel primario para una gestión eficaz de la calidad del software. En la Figura 12 se muestra la interfaz de usuario de la herramienta SQUORE.

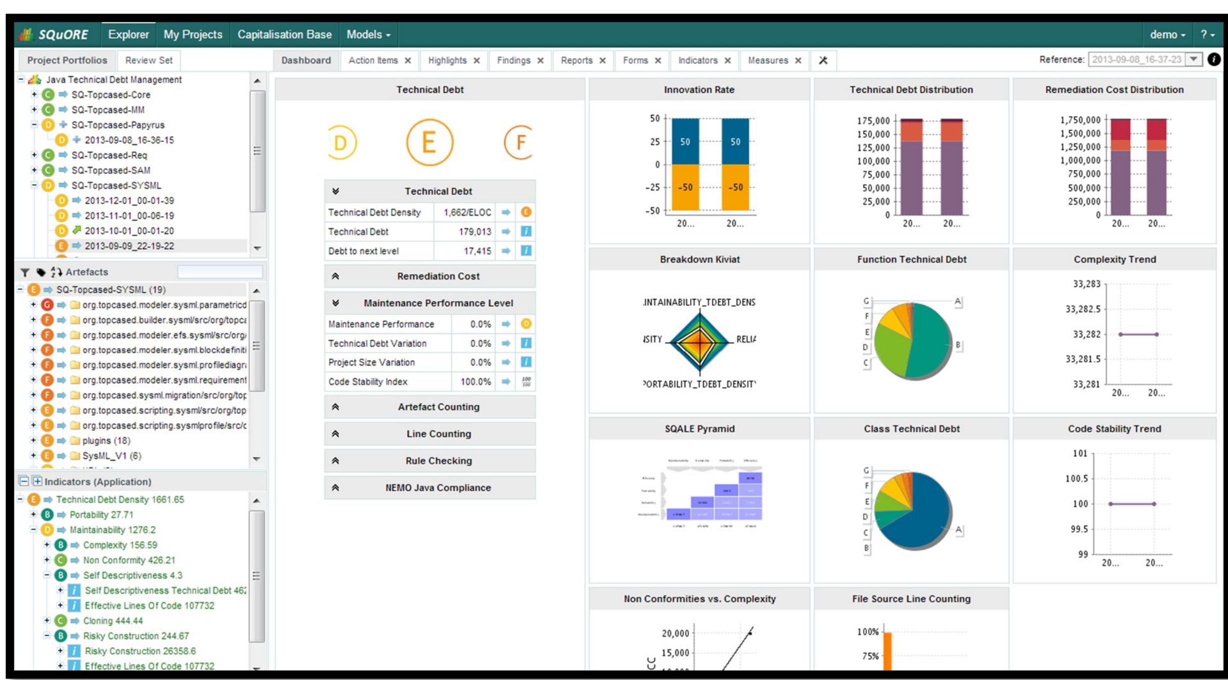


Figura 12.Dashboard de herramienta SQUORE.

Fuente: (“Squoring,” n.d.)

Características de la herramienta:

- Posee un sistema de calificación, fácil de entender, en el que proporciona información precisa sobre estimaciones de la deuda técnica, que puede ser en días, hombre o moneda.

[13]http://www.squoring.com/en/produits/tableau-de-bord-squore/

- Existen algunos widgets adicionales que proporcionan información sobre la deuda técnica de un proyecto.
- Analiza el software mediante las siguientes métricas: complejidad, comentarios, duplicaciones, dependencias, tamaño de software.
- Realiza análisis de deuda técnica.
- Su método de análisis es el estático.

- **Kiuwan**[14]

Es una herramienta de análisis estático de código, dedicada a la analítica de las aplicaciones software, así como a la medición de la calidad y medida de seguridad del código fuente de un proyecto, el cual se lo puede hacer mediante una pequeña aplicación descargable, o en la nube, subiendo el código a la propia plataforma. La Figura 13 se visualiza un ejemplo de resultados que proporciona la herramienta Kiuwan.

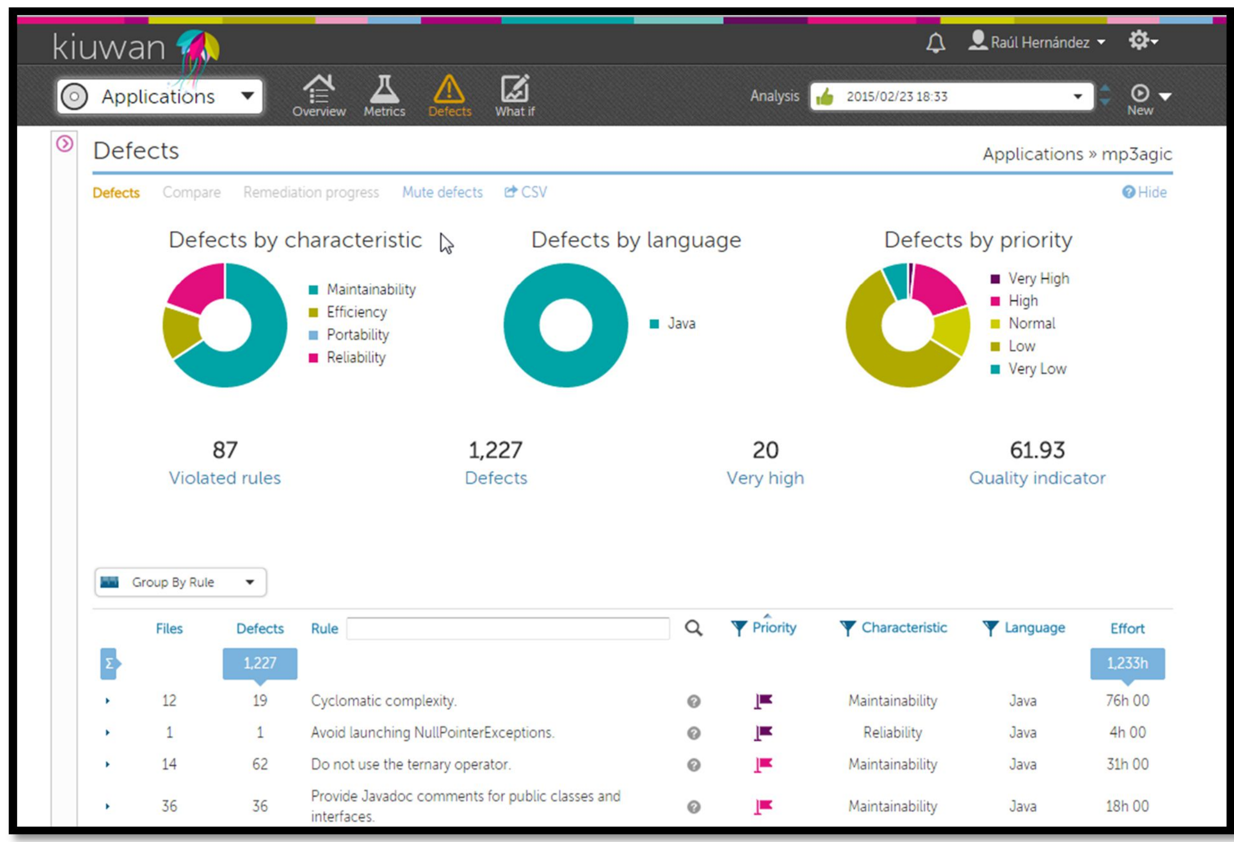


Figura 13.Dashboard de herramienta Kiuwan.

Fuente: (“Kiuwan,” n.d.)

[14]https://www.kiuwan.com/es/

Características de la herramienta

- Permite evaluar la deuda técnica para mejorar la toma de decisiones.
- Analiza métricas como: complejidad, duplicaciones, tamaño de código, acoplamiento, dependencias.
- Se puede realizar el análisis mediante un analizador web o también por medio de un analizador local que proporciona la herramienta.
- Es una herramienta colaborativa.
- Generar planes de acción que te ayuden a conseguir tus objetivos con el mínimo riesgo

- **PMD[15]**

PMD (Programming Mistake Detector) es una herramienta de análisis estático para el código Java. Es un programa gratuito y se encuentra disponible para sistemas operativos Windows, Mac OS X y Linux. Además incluye un conjunto de reglas integradas y admite la capacidad de escribir reglas personalizadas. Es posible la integración de la herramienta con un IDE que soporte PMD. La Figura 14 presenta un ejemplo de reporte tras realizar un análisis de calidad con PMD; en el que indica el número de violaciones por métrica y la ubicación de cada una.

Summary

Rule name	Number of violations
CommentRequired	193
DefaultPackage	23
ConfusingTernary	3
MissingSerialVersionUID	3
NullAssignment	5
UnusedImports	12
AvoidDuplicateLiterals	1
PreserveStackTrace	3

Detail

PMD report

Problems found

#	File	Line	
1	capaDatos\Conductor.java	40	fieldCommentRequirement Required
2	capaDatos\Conductor.java	43	fieldCommentRequirement Required
3	capaDatos\Conductor.java	47	fieldCommentRequirement Required
4	capaDatos\Conductor.java	51	fieldCommentRequirement Required

Figura 14. Ejemplo de reporte de análisis de software presentado por PMD

Fuente: El Autor

[15] https://pmd.github.io/

Características de la herramienta:

- Trabaja con los siguientes lenguajes de programación: Java, C, C++, C#, Ruby.
- Analiza métricas como: duplicaciones, complejidad, tamaño de código,.
- Se adapta a SonarQube mediante un plugin gratuito.
- No realiza análisis de deuda técnica.
- Técnica: Análisis estático.

- **Mia-Quality**[16]

Es una herramienta de análisis estático ayuda a identificar acciones en el código fuente para ayudar a reducir la deuda técnica de aplicaciones. Mia-Quality es compatible con las normas de calidad ISO9126 y también implementa el método SQALE. La Figura 15 muestra un ejemplo de los resultados de análisis estático que presenta.

Figura 15. Proceso de análisis de calidad de software con Mia-Quality.

Fuente: El Autor

Características de la herramienta:

- Trabaja con los siguientes lenguajes de programación: Java, C#, .NET, COBOL.
- La capacidad de elevar el nivel de madurez del software.
- Facilidad de integración de nuevos lenguajes, nuevas medidas
- La integración nativa con la calidad del portal de sonarqube

[16] http://www.mia-software.com/en/produits/mia-quality/

- Analiza métricas como: duplicaciones, complejidad, tamaño de código.
- No realiza análisis de deuda técnica.
- Técnica: Análisis estático.

De las herramientas revisadas en la presente sección todas realizan un análisis estático para detectar errores en el software, pero herramientas como: Understand, NDepend y SourceMeter, poseen métricas básicas que no permiten generar un análisis completo al software. Además las herramientas que permiten la identificación de DT son: SonarQube, Kiuwan y Squore. Es por ello que es necesario basarse en las métricas que analiza cada herramienta antes de elegir una adecuada para realizar un análisis de calidad al software.

1.8 Modelos de calidad para la identificación de deuda técnica.

- En base a las herramientas de análisis de calidad de software, éstas por lo general utilizan métodos y estándares de calidad para la adopción en el proceso de identificación de la deuda técnica; mediante el análisis de las normas de calidad de software. A continuación se exponen algunos modelos para la identificación de deuda técnica, entre las que constan ISO/IEC 25010, ISO/IEC 9126, SQALE, SIG/TÜViT, Design Structure Matrix.

- **ISO / IEC 25010**

Este modelo determina características de calidad que se deben considerar a la hora de evaluar las propiedades de un producto software. Los atributos de calidad recogidos que se consideran en peligro cuando se incurre en la deuda técnica, utilizando el modelo de calidad de los productos software propuesto en la norma ISO / IEC 25010, son: portabilidad, capacidad de mantenimiento, seguridad, confiabilidad, usabilidad, entre otros(Li et al., 2015).

Algunas de las características de este modelo se derivan de los de ISO 9126 y es aplicable a productos de software y sistemas informáticos.

- **ISO /IEC 9126**

Este estándar clasifica la calidad del software en varias características, que son: funcionalidad, fiabilidad, usabilidad, mantenibilidad, eficiencia, portabilidad y calidad en uso.

En un estudio del autor (Alzaghoul, 2013), se realizó un análisis estático en algunas aplicaciones, los cuales se evaluaron, después dela revisión de la norma ISO / IEC 9126; en

la que los resultados fueron ponderados por su severidad, y los factores de salud detectados fueron escalabilidad, capacidad de respuesta, transferibilidad y mutabilidad de aplicaciones.

Además el modelo ISO / IEC 25010 es la última revisión de la norma ISO / IEC 9126.

- **SQALE**

El método SQALE (Evaluación de la Calidad de Software Basado en el ciclo de vida de las expectativas), se utiliza para estimar la calidad y la deuda técnica del código fuente del software (J. L. Letouzey, 2012), en el que se organizan los requisitos no funcionales del software relacionados con las características de calidad que propone este método, las cuales son: mantenibilidad, testeabilidad, fiablidad, portabilidad, reusabilidad, eficiencia y seguridad (J. L. Letouzey & Ilkiewicz, 2012).

Los modelos de calidad y análisis SQALE se han utilizado para realizar muchas evaluaciones de código fuente del software, de diversos tamaños y en diferentes dominios de aplicación (J. L. Letouzey & Coq, 2010).

- **SIG/TÜViT**

Este método está destinado a la evaluación estandarizada y certificación de la calidad técnica del código fuente de productos de software. El alcance de sus principales métricas y criterios de evaluación se limitan a la calidad interna de mantenimiento y sus sub-características incluyendo: análisis, modificación, la capacidad de prueba, la modularidad y la reutilización (Visser, 2014).

Los criterios de evaluación SIG / TÜViT están dirigidos a la calidad técnica del código fuente de los productos de software, donde la calidad técnica significa la característica de calidad interna de la capacidad de mantenimiento y sus sub-características de acuerdo con la norma ISO / IEC 25010. Estas sub-características son análisis, modificación, la capacidad de prueba, la modularidad y la reutilización.

- **Design Structure Matrix (DSM)**

Este método se enfoca en los servicios web y el cambio hacia los servicios basados en la nube, en el cual se presenta un método que construye en la estructura de la matriz de diseño

(DSM) e introduce tiempo y la complejidad costo de propagación métricas de costo para evaluar el valor de las decisiones de aplazamiento en relación con los cambios en la estructura. DSM muestra las dependencias entre los componentes de la arquitectura (Alzaghoul, 2014).

En base a los conceptos expuestos en el presente capitulo que son la pilar de conocimiento para el desarrollo del presente trabajo de fin de titulación, se identifican tres fases de análisis de software para la gestión de la deuda técnica, que son: identificación, medición y monitoreo, de las cuales se utilizara la identificación y la medición de la deuda técnica para evaluar aplicaciones de software que contengan fallas a nivel de código y diseño (tomando como punto de referencia su código fuente o ejecutable), mediante el uso de herramientas de análisis estático expuestas en la Tabla 5 y un método de calidad de software que se enfoque al análisis de DT,

CAPITULO 2: PROCESO DE IDENTIFICACIÓN Y MEDICIÓN DE LA DEUDA TÉCNICA

Para llevar a cabo la gestión de la deuda técnica sobre un proyecto de software de manera correcta, es preciso utilizar correctamente sus actividades, para una buena toma de decisiones ante la detección de problemas que comprometen la calidad de software.

Actividades de identificación y medición permiten gestionar la deuda técnica. La identificación se encarga de detectar los daños que se han provocado a un sistema de software o a parte del mismo; debido a decisiones intencionales y no intencionales que se toman dentro de los equipos técnicos de desarrollo. Mientras que la medición de la deuda técnica se la realiza para poder estimación del esfuerzo necesario para poder reparar los componentes del sistema a los que se les haya detectado defectos en el software (Kuipers, 2011).

2.1 Identificación y medición de la deuda técnica.

La identificación es una actividad de la gestión de la deuda técnica que detecta los daños provocados al software. Esta actividad se la puede automatizar mediante el uso de alguna herramienta de análisis de calidad de software (ver Figura 16), su proceso es como sigue:

- Inicialmente se toman como entrada los diferentes archivos del software en el lenguaje que han sido codificados (Java, PhP, Cobol, C++, etc).

- Seguidamente se analizan los datos de entrada a través de una herramienta de análisis de calidad de software; en el que se ingresa los archivos del código fuente (.java, .php. SQL, etc) para procederlos a evaluar en base a reglas de programación correspondientes al lenguaje de codificación del software.

- Finalmente posterior al análisis se presentan las violaciones detectadas a través de un informe, en el cual se indican las reglas que se han violado y la característica de calidad a la que corresponde cada una, esto se basa a la norma de calidad que se esté utilizando.

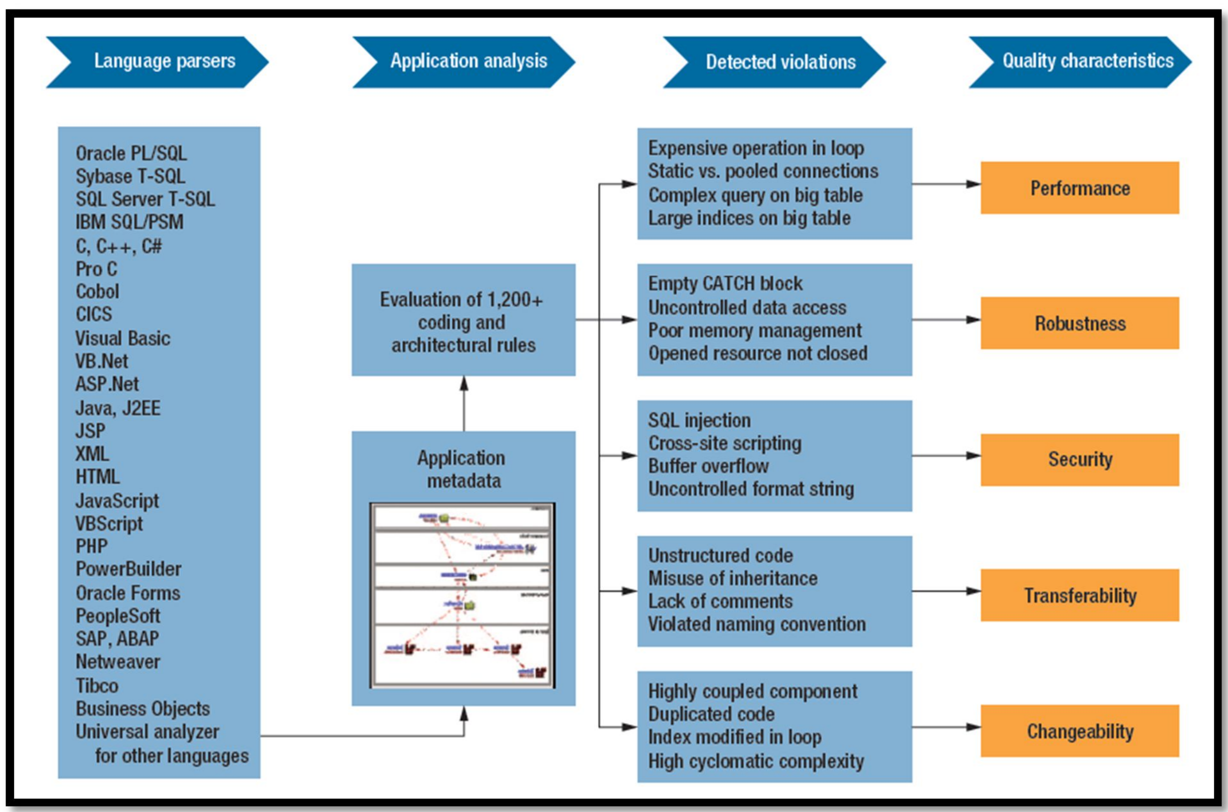


Figura 16.Proceso de identificación de deuda técnica a nivel de código.

Fuente: (Curtis et al., 2012a)

En la cuantificación o medición de la DT se estima es el esfuerzo de reparación del software a un nivel que se considere ideal. Existen dos factores que se deben tener en cuenta en la estimación, que son la fracción de reconstrucción y el valor de la reconstrucción (Kuipers, 2011).

- **Fracción de reconstrucción.-** Es una estimación del porcentaje de líneas de código que necesitan ser cambiadas para mejorar la calidad del software.

- **Valor de la reconstrucción.-** Es una estimación del esfuerzo que puede ser en unidad de tiempo, el cual será gastado para reconstruir un sistema.

Se puede asemejar el factor de reconstrucción como la identificación de la DT y el valor de reconstrucción con la medición de la DT, ya que la fracción de reconstrucción identifica el daño que posee el software y el valor de reconstrucción es la cantidad se esfuerzo de remediación de la deuda técnica; este se calcula en base a la cantidad de daño del software, el cual se obtiene en el proceso de identificación. En la Figura 17 se observa un proceso en el que se complementan ambas actividades, las cuales trabajan mediante el uso de

herramientas de análisis de calidad de software, las cuales facilitan los procesos de identificación y medición.

- Inicialmente se recogen los diversos conjuntos de datos del proyecto, en él se incluye el historial del repositorio GIT[17](software de control de versiones) de un proyecto. También se utiliza como entrada un conjunto de violaciones de código detectadas tras un análisis previo a través de la herramienta Understand.

- A continuación se utiliza la herramienta Titan[18] para detectar puntos de daños arquitectónicos

- Seguidamente los daños arquitectónicos encontrados se proceden a evaluarse por arquitectos de software para entender si los problemas que se han identificado son reales, y vale la pena tratar.

- Finalmente, para cuantificar la deuda arquitectónica por los arquitectos, se necesitan datos adicionales del proyecto como el número de líneas de código para realizar la estimación de deuda técnica, la cual sirve de base para la toma de decisiones sobre la corrección de los problemas del software.

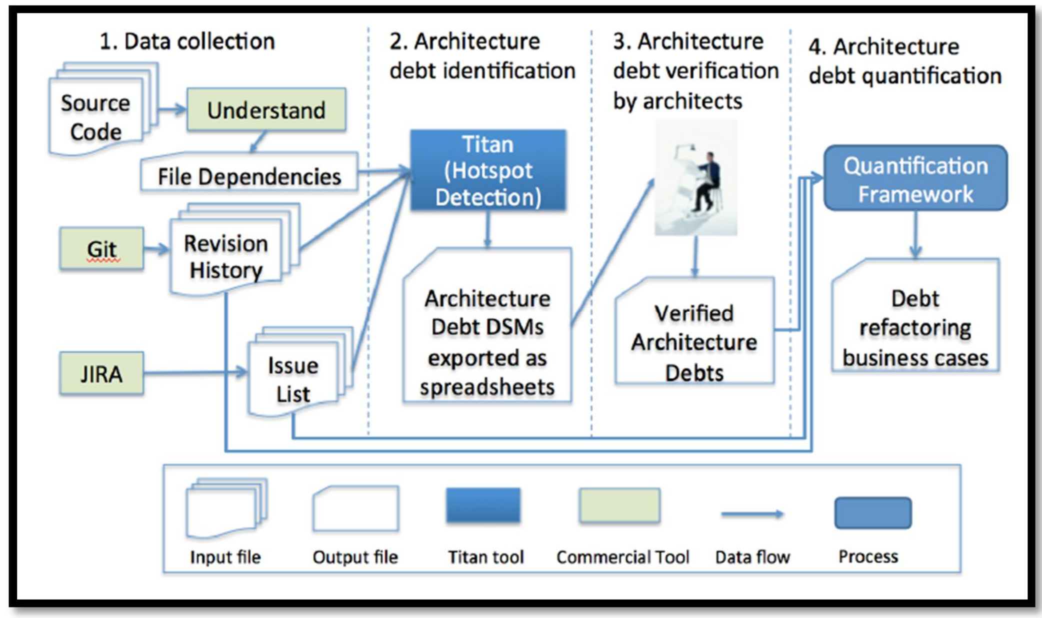


Figura 17.Proceso de identificación y medición de deuda técnica a nivel de arquitectura.

Elaboración:(Kazman et al., 2015)

[17]https://git-scm.com/
[18] http://www.titaniisoftware.com/home

Dado los procesos identificación y medición de deuda técnica (los mismos que se aplican a nivel de código y/o diseño) que se revisa en la Figura 16 y Figura 17, se proponen los siguientes pasos para la valoración de un sistema de software a nivel de código y diseño, los cuales se muestran en la Figura 18.

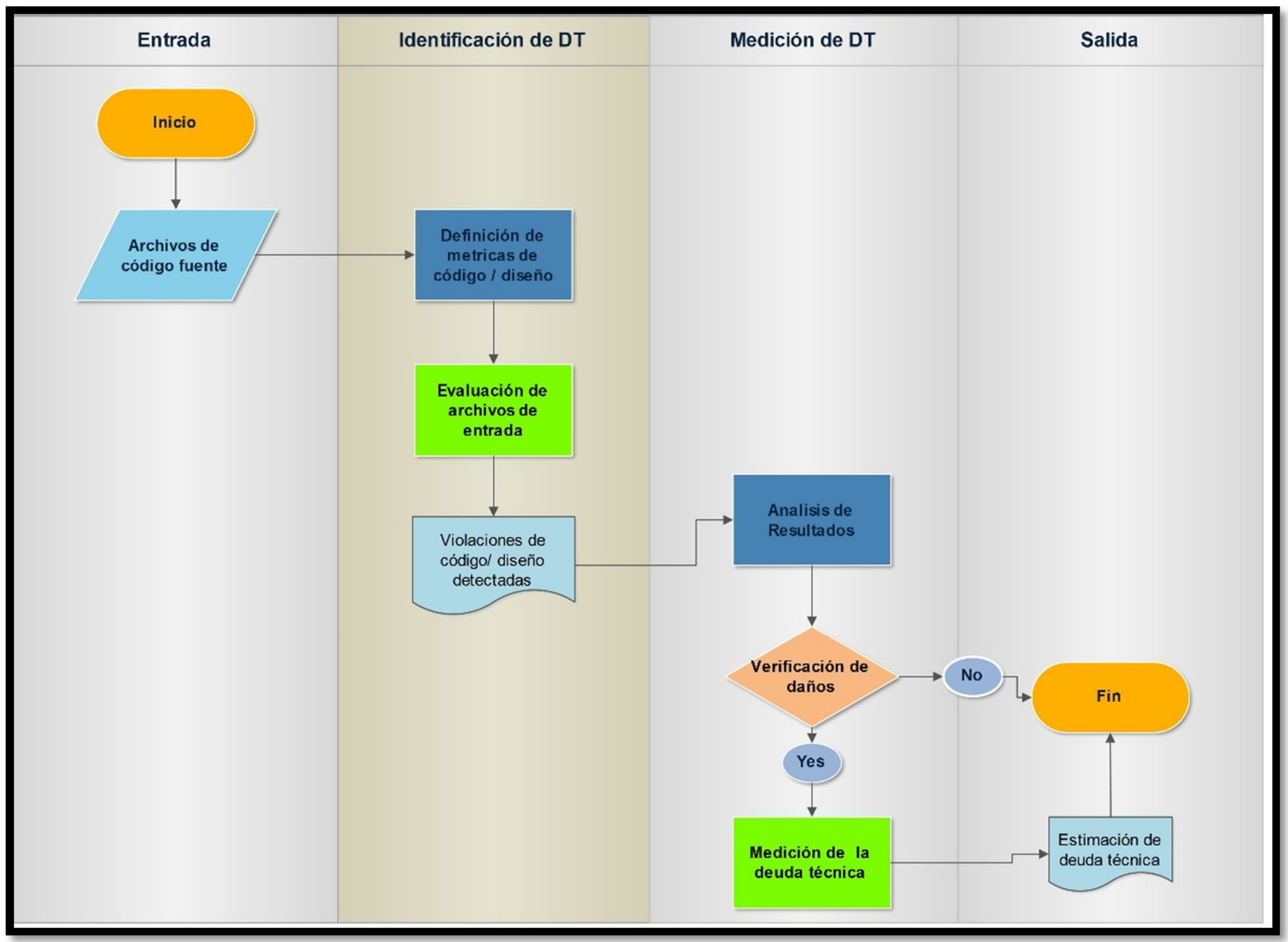


Figura 18.Proceso de Gestión de deuda técnica.

Elaboración: El autor

- **Entrada**

  Como primer paso, se debe recoger los diversos conjuntos de datos del proyecto (archivos. Java, C/C#, PHP, JavaScript, etc), es decir el código fuente que se va a evaluar, como: *.class* archivos en Java, *.dll* archivos en C #, etc.

- **Identificación de DT**

  Seguidamente, definen las métricas que se van a evaluar sobre el sistema objeto de análisis, las métricas pueden ser líneas de código (LOC), la duplicación de código, la complejidad ciclomática, nivel de dependencia los cuales se recogen de los elementos del sistema a evaluar y se calcula las zonas afectadas de arquitectura dentro del código fuente.

- **Medición de DT**

  Como tercer paso, los resultados de las métricas obtenidas a partir del análisis de la deuda arquitectónica son analizados mediante un grupo de arquitectos con lo cual se evalúa si es necesario proceder a una corrección del sistema de software o partes del mismo. En caso de ser necesario corregir el software se cuantifica la deuda técnica arquitectónica mediante la estimación de esfuerzo requerido para refactorizar las fallas de arquitectura que se ha identificado.

- **Salida**
  Finalmente, se presentan los resultados de la DT del software evaluado, los mismos que utilizaran el equipo de mantenimiento para la corrección del software.

Para el cálculo de la deuda técnica de manera formal, se requiere de un método que permita determinar el nivel "ideal" de calidad de un sistema de software. El método que se va a utilizar en el presente trabajo es SQALE, ya que, es un método genérico que se adapta a cualquier proyecto y se enfoca al cálculo de la deuda técnica.

2.1.1 Método SQALE.

El modelo de análisis de calidad SQALE[19] (Software Quality Assessment based on Lifecycle Expectations) proporciona orientación y apoyo para mejorar la calidad del código fuente, mediante la implementación del concepto de deuda técnica.

Dentro de las características que posee el modelo SQALE, se destacan las siguientes:

[19]http://www.sqale.org/

- Se integra fácilmente a herramientas como: SonarQube, Squoring, Mia-software.
- Se basa en norma ISO 9126.
- Ayuda a la evaluación y medición de la deuda técnica.
- Mide la deuda técnica en días.
- Es de libre acceso.
- Cada herramienta que utiliza este método lo implementa a la manera de cada una.

La calidad del código de un software se relaciona con los requisitos no funcionales, ya que estos son los que especifican la calidad del producto, y este método organiza los requisitos como criterios para la evaluación del mismo. Los atributos que se consideran en este modelo se organizan en tres niveles jerárquicos características, subcaracterísticas, requisitos el código fuente:

- **Características.-** Son ocho "habilidades" utilizadas de la norma ISO 9126, ya que estas afectan las actividades típicas de ciclo de vida de una aplicación de software, estas actividades se presentan en la Figura 19.

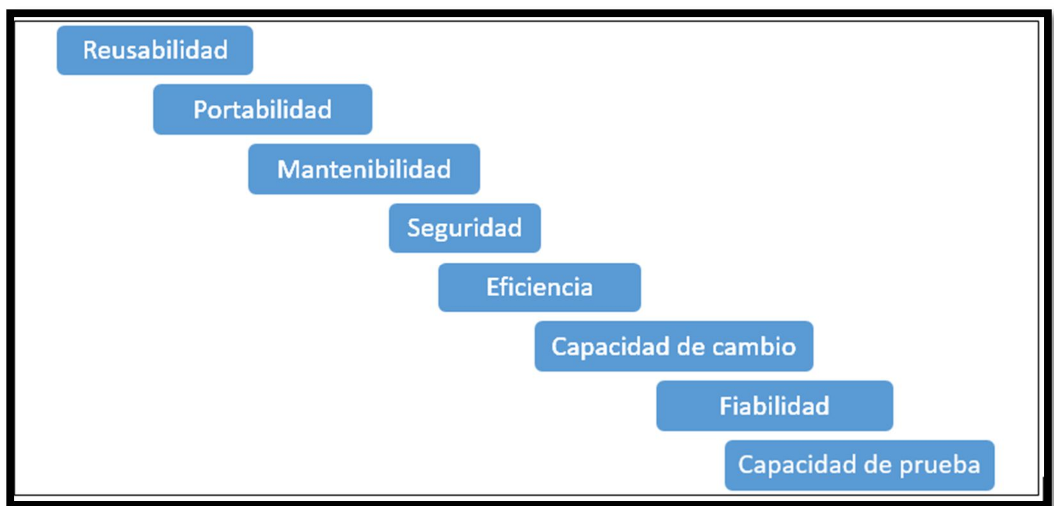


Figura 19.Características de calidad de SQALE.

Elaboración: El autor

- **Subcaracterísticas.-** Se utilizan para agrupar los requisitos y así permitir realizar análisis con varios niveles de abstracción. Algunos de ellos pueden suprimirse si corresponden a una actividad no es aplicable en el contexto del software que se está evaluando.

- **Requisitos del código fuente.-**Son los requerimientos que se toman antes de empezar con el desarrollo de un producto software y que se toman como referencia para conocer si el sistema final cumple con los objetivos del proyecto.

El método se basa en nueve principios y cuatro conceptos (J. L. Letouzey & Ilkiewicz, 2012), con el fin de obtener una estimación precisa de la calidad y de la deuda técnica.

2.1.1.1 Principios de SQALE.

El método de calidad de software consta de los 9 principios fundamentales en el que se basa para la medición de calidad de software (J.-L. Letouzey, 2012). Estos 9 principios fundamentales se indican a continuación.

- La calidad del código fuente es un requisito no funcional.
- Los requisitos en relación con la calidad del código fuente tienen que ser formalizadas de acuerdo con los mismos criterios de calidad como cualquier otro requisito funcional.
- La evaluación de la calidad del código fuente es la distancia entre su estado actual y el objetivo de calidad esperado, considerando que el objetivo es el cumplimiento de los requisitos de calidad de código fuente.
- El Método SQALE evalúa la distancia a la conformidad con los requisitos teniendo en cuenta el costo de remediación necesaria de lo que el código fuente de la conformidad.
- El Método SQALE evalúa la importancia de una no conformidad, considerando los costos resultantes de la entrega del código fuente con esta falta de conformidad.
- El método SQALE respeta la condición de representación. El Método SQALE ha sido diseñado por el respeto de esta condición (ver una definición de esta condición en el apéndice). Esto afecta a la elección de los requisitos, su organización en el modelo de calidad y las normas de adición.
- El método SQALE utiliza además para la agregación de los costos de remediación, los gastos no relacionados con la remediación y para el cálculo de los indicadores.
- El método de SQALE es ortogonal, es decir, que un requisito relacionado con uno de los atributos internos del código aparece sólo una vez en el Modelo de Calidad. Un requisito está vinculado a una sola subcaracterística de calidad.
- Modelo de Calidad del Método SQALE toma ciclo de vida del software en cuenta. Características, subcaracterísticas y requisitos del modelo de calidad del Método

SQALE están organizados de tal manera que refleje la cronología de las necesidades tal como aparecen en el ciclo de vida del software.

2.1.1.2 Estructura SQALE.

El método se basa en cuatro conceptos para manejar correctamente la gestión de la deuda técnica, los cuales se denominan: modelo de calidad, modelo de análisis, índices e indicadores.

2.1.1.3 Modelo de calidad

Este modelo forma parte de la identificación de la deuda técnica y es utilizado para la formulación y organización de los requisitos no funcionales del software. Se organizan las características con las sub-características que se evalúan. Las sub-características se pueden suprimir no corresponden a una actividad no aplicable en el contexto del software que se está evaluando.

2.1.1.4 Modelo de análisis

Este modelo pertenece a la medición de la deuda técnica, en el que se analiza los resultados de las violaciones detectadas tras la evaluación de calidad del software, transformándolas en costos de remediación y no remediación.

- **Costos de remediación.-** Se refiere al coste que se debe pagar corregir las violaciones del software y se lo define a cada regla de programación de forma individual, y en caso de utilizar una herramienta de análisis este costo de remediación ya viene definido por defecto para cada regla de programación, por ejemplo los costos se pueden clasificar en cinco tipos, como se muestra en la Tabla 6.

Tabla 6. Tabla de costos de remediación

Tipo	**Descripción**	**Ejemplo**	**Costo de remediación**

Tipo 1	Corregible con una herramienta	Cambio de carácter.	1
Tipo 2	Remediación Manual , pero no influye	Agregue un poco de comentarios	5
Tipo 3	Impacto local , sólo necesita las pruebas unitarias	Reemplazar una instrucción por otra	10
Tipo 4	Impacto medio , necesitan pruebas de integración	Cortar una gran función en dos	20
Tipo 5	Gran impacto , necesitan una validación completa	Cambie dentro de la arquitectura	30

Elaboración: El autor

- **Costos de no remediación.-** Es el costo que una organización debe pagar por no solucionar una violación a un requerimiento no funcional o de calidad, el cual se lleva a cabo desde el punto de vista de negocio o propietario del producto y se lo define mediante una lógica que consiste en la asociación de una criticidad para cada requisito que no se cumpla, por ejemplo los costos de no remediación se pueden clasificar en cinco tipos, como se muestra en la
- Tabla *7*.

Tabla 7. Tabla de costos de no remediación

Tipo	**Descripción**	**Ejemplo**	**Costo de remediación**
Bloqueante	Será o puede dar lugar a un bug	División por cero	30
Critico	Tiene un impacto alto / directo en el costo mantenimiento	Copiar y pegar	20
Mayor	Tendrá un impacto potencial / impacto medio sobre el costo de mantenimiento	Lógica compleja	10

Menor	Tendrá un bajo impacto en el costo mantenimiento	Convenio de denominación	5
Informe	Muy bajo impacto , es sólo un informe de costos de remediación	Tema de Presentación	1

Elaboración: El autor

El valor total de la deuda técnica es la suma de los costos de remediación, que representa al coste que se debe retribuir para corregir los incumplimientos detectados en el software (frente a los requisitos de modelo).

2.1.1.5 Los índices

Los índices son variables que representan costos por cada característica de calidad utilizada en el modelo SQALE; unos representan la deuda técnica de la aplicación y los otros se usan para analizar la deuda y se miden en relación a una misma unidad, sea monetaria, de trabajo o una unidad simbólica.

Para cada característica de calidad del método SQALE se definen los siguientes índices:

- Índice de Capacidad de prueba SQALE: STI
- Índice de fiabilidad SQALE: SRI
- Índice de Variabilidad SQALE: SCI
- Índice de Eficiencia SQALE: SEI
- Índice SQALE de seguridad: SSI
- Índice de Capacidad de mantenimiento SQALE: SMI
- Índice de Portabilidad SQALE: SPI
- Índice de Reutilización SQALE: SRuI
- El Índice de Calidad SQALE: SQI, es el que mide la deuda técnica.

El valor de la deuda técnica es el resultado de la suma de los costos de remediación de todas las características del modelo de calidad, la cual se define con el índice SQI.

Estos índices se usan en los indicadores dela deuda técnica como el indicador "SQALE Pirámide". Para calcular el valor de deuda técnica por cada característica se aplica la (Ecuación *1*).

$$C = \sum_{SC=1}^{SC=n} NoViolaciones\ x\ Costo_remediación$$

(Ecuación 1). Valor de deuda técnica por característica

Elaboración: El autor

Donde C es la característica, la cual se debe definir con su índice correspondiente, SC la subcaracteristica que pertenece a la característica principal, NoViolaciones es el número de violaciones y Costo_remedeciacion es el tiempo de remediación asociado a cada regla.

EL valor total de la deuda técnica de un proyecto se la calcula aplicación la (Ecuación 2), que es la suma del cálculo de las características a evaluar.

$$DT = \sum_{C=1}^{C=n} C$$

(Ecuación 2). Valor total de deuda técnica.

Elaboración: El autor

Donde DT es el total de la deuda técnica y C es el valor de la característica calculada.

También existen los índices (Ver Tabla 8) consolidados que se utilizan para la descripción sintetizada de la calidad de un proyecto.

Tabla 8. Índices consolidados

Ecuación	**Índice**
SCTI = STI	(Ecuación 3). Índice Comprobabilidad
SCRI = STI + SRI	(Ecuación 4). Índice Consolidado Confiabilidad
SCMI = STI + SRI + SCI.	(Ecuación 5). Índice Consolidad Mantenibilidad
SCCI = STI + SRI + SCI	(Ecuación 6). Índice Capacidad de Cambio

Elaboración: El autor

Fuente: (J. L. Letouzey, 2012)

Las Ecuaciones Elaboración: El autor),**¡Error! No se encuentra el origen de la referencia.**),(Ecuación *5*),) definen las características de calidad que son necesarias efectuar para cumplir otra característica, por ejemplo si se desea que su el código sea fiable, se debe incluir en su Modelo de Calidad un requisito relacionado con la cobertura de código de prueba.

2.1.1.6 Los indicadores

El método define algunos indicadores importantes, aunque se pueden definir más según la situación. Estos indicadores se han definido para proporcionar una representación visual de la deuda técnica, entre los más conocidos son:

- **Clasificación.-** Esta se representa a través de cinco valores que son: A, B, C, y D. Esta calificación que se les da a una característica concreta y un artefacto concreto es el resultado de una comparación del coste estimado de corregirla respecto a la estimación del coste de desarrollo de ese artefacto, o lo que es lo mismo la deuda técnica de tu artefacto software.

Rating	Up to	Color
A	1%	
B	2%	
C	4%	
D	8%	
E	∞	

Figura 20. Calificación de SQALE

Elaboración: (J.-L. Letouzey, 2012)

La Figura 20 representa el nivel de daño mediante colores, siendo el verde como un buen resultado y el rojo como la detección de muchos errores y mediante estos colores se visualiza la clasificación SQALE de un proyecto de software; esta clasificación se genera mediante el **ratio de la deuda técnica**, que es una métrica de la misma que indica la ratio entre la deuda técnica actual y el esfuerzo que se debería de invertir para reescribir todo el código desde el principio. Y esta medida se la calcula mediante la (Ecuación 7).

$$\text{Ratio de DT} = \frac{\text{deuda_técnica}}{\text{coste_estimado de desarrollo}}$$

(Ecuación 7). Ratio de la deuda técnica

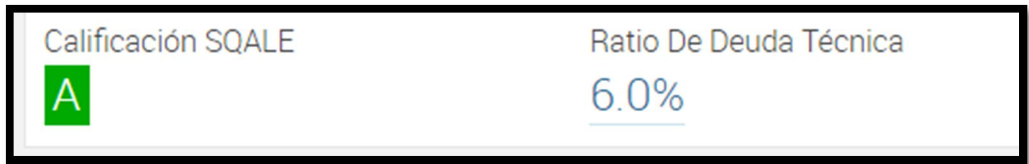


Figura 21. Resumen de da DT de un proyecto

Elaboración: El autor

Tabla 9. Escala de categorización del ratio SQALE

Ratio	Calificación
<10%	A
10% - 20%	B
21% - 50%	C
51% - 100	D
>100%	E

Elaboración: El autor

La Figura 21 presenta la calificación SQALE y el ratio de la deuda técnica, los cuales transmiten de forma breve la calidad de un proyecto dándole una categoría entre la A y la E siguiendo la escala que se presenta en la Tabla 9, en donde, se indica que mientras mayor es el ratio, mayor es la complejidad del proyecto.

- **Pirámide**

Es una pirámide figurativa que representa los índices consolidados como se indica en la Figura 22, y se la interpreta de forma bottom up, donde la parte más pequeña es la que tiene asociado el coste más alto, es decir es la más importante. La capacidad de pruebas está como principal (Testability), ya que, primeramente hay que asegurarse que una aplicación pueda probarse y luego asegurar el resto de características hacia arriba.

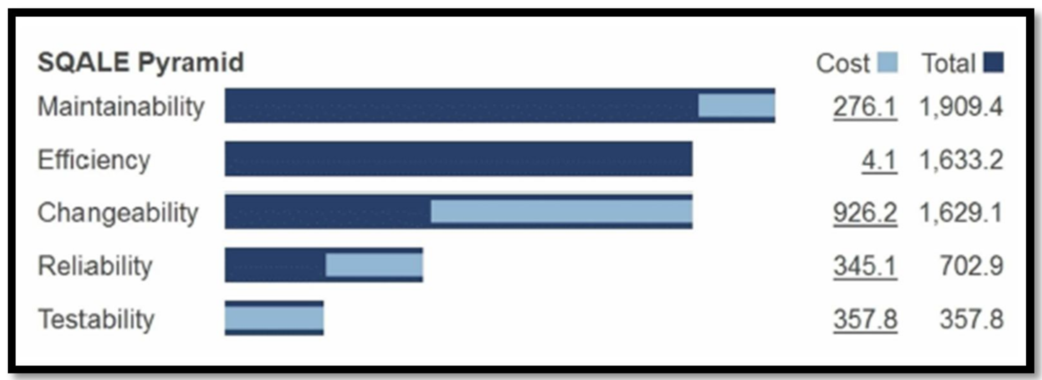


Figura 22.Ejemplo de pirámide SQALE de la herramienta SonarQube.

Elaboración: (J. L. Letouzey, 2012)

La pirámide de la Figura 22 presenta el tiempo de solución de problemas por característica, el cual se calcula mediante los índices consolidados, según el ejemplo la característica de confiabilidad demora 345 días en remediar los daños existentes, y esto es desde un punto analítico, pero basándose en la realidad y tomando en cuenta algunos inconvenientes como métodos muy complejos, los días de reparación son de 702.

2.2 Medidas de cálculo de la deuda técnica.

El método SQALE como parte de solución al problema de calidad de software, define factores de calidad resultado de la ISO 9216, que se deben evaluar en un sistema de software, es por ello que en la Tabla 10 se describe la distribución de las características, subcaracterísticas y las métricas a nivel de código y diseño que evalúa el método SQALE.

Tabla 10. Ejemplo de características y subcaracterísticas de método SQALE.

Característica	Subcaracteristicas	Requerimientos	Descripción
Reusabilidad	Complejidad	- Complejidad de archivos - Complejidad de métodos	Se considera al uso de componentes de software, durante la construcción de nuevos sistemas de software.
Portabilidad	Portabilidad relacionada al lenguaje	- Dependencia de codificación por defecto	Es la capacidad del producto para migrar de un entorno a otro.

Mantenibilidad	Comprensibilidad	- No usar doble puntero. - No usar declaraciones(goto , break fuera de un switch)	Es la capacidad del producto software para ser modificado, estas modificaciones pueden ser mejoras o correcciones, se incluye la codificación, el diseño y documentación de cambios.
	Legibilidad	- Inicio de un nombre de variable con letra minúscula.	
Seguridad	Seguridad relacionadas a sentencias	- Cookies HTTP - Conexiones con base de datos con passwords vacíos	Es la capacidad que tiene el producto para proteger información de personas o de sistemas que no estén autorizados.
	Seguridad relacionadas al usuario	- Encapsulación	
Eficiencia	Eficiencia relacionada a la RAM	- Variables no utilizadas, parámetros o constante en el código. - Bucles infinitos	Capacidad del producto para funcionar correctamente utilizando de mejor manera los recursos bajo situaciones determinadas
Capacidad de cambio	Complejidad	- Complejidad ciclomática - Complejidad de archivos - Complejidad de métodos - Relación de líneas de comentarios con líneas de código	La capacidad del producto de software para permitir una modificación especifica que se aplicará.
Capacidad de cambio	Complejidad	- Complejidad ciclomática - Complejidad de archivos - Complejidad de métodos - Relación de líneas de comentarios con líneas de código	La capacidad del producto de software para permitir una modificación especifica que se aplicará.
	Documentación	- Líneas de comentarios - Densidad de comentarios - Líneas de código comentadas.	

Fiabilidad	Fiabilidad relacionada a la lógica	- Mala declaración de clases - Elementos inutilizados	Se refiere a la precisión con la que el producto funciona de acuerdo a los requerimientos preestablecidos.
	Tolerancia a fallos		
Capacidad de pruebas	Pruebas unitarias	- Todos los módulos son accesibles. - Número de parámetros en una llamada módulo <6	Es el grado en que el producto ha sido comprobado.

Elaboración: El autor

Como parte de entrada de datos para la identificación de la DT se requiere la organización de las características de calidad del método SQALE con los requisitos del proyecto, para conocer cuáles son las métricas específicas que se van a evaluar.

2.2.1 Características SQALE a evaluar.

El método menciona ocho características que se utilizan para evaluar el software según el nivel de madurez en el desarrollo del proyecto y de acuerdo a la documentación del método SQALE, es posible no considerar ciertas características en la evaluación de un software aplicando el presente método, es por ello que para el presente trabajo se eligieron las siguientes: **Mantenibilidad, Confiabilidad, Capacidad de cambio y Capacidad de prueba**, debido a que estas características contienen las reglas a nivel de código y diseño que comúnmente se pasan por alto.

En la Tabla 11 se definen las medidas generales que se van a utilizar en el análisis de deuda técnica del presente trabajo, las cuales se adaptaran según el lenguaje que se va a analizar.

Tabla 11. Selección de subcaracterísticas y reglas para el análisis de deuda técnica.

Característica	Subcaracterística	Regla	Tipo de DT
Mantenibilidad	Comprensibilidad	Densidad de líneas de código	Código
	Comprensibilidad	Clases no deben ser muy complejas	Código
	Comprensibilidad	Nombres de variables	Código
	Legibilidad	No debe haber líneas de código demasiado largas.	Código
	Legibilidad	Nombres de clases claras	Código

	Comentarios		Código
Confiabilidad	Lógica	Evitar duplicaciones de estructuras de decisión	Código
	Tamaño de código	Complejidad ciclomática	Código
	Arquitectura	Módulos sin uso	Diseñó
	Arquitectura	No usar métodos sin uso	Diseño
Capacidad de cambio	Arquitectura	Evitar ciclos de paquetes	Diseño
	Arquitectura	Evitar acoplamiento de clases	Diseño
	Lógica	Bloques duplicados	Código
Capacidad de prueba	Nivel de unidad	Métodos demasiado complejos	Código
	Capacidad de prueba	Expresiones no deben ser demasiadas complejas	Código
	Capacidad de prueba	Evitar clases demasiadas complejas	Código
	Capacidad de prueba	Evitar herencia muy profunda.	Código

Elaboración: El autor

Finalizando este capítulo se concluye que para la realización de los procesos de identificación y medición de la deuda técnica se implementara método SQALE al análisis de DT, ya que al ser un método genérico puede adaptarse a cualquier proyecto y además garantiza una correcta estimación de resultados, es por ello que para la realización del presente trabajo en la Tabla 11 se definen las caracterizas de calidad a evaluar, en donde la mantenibilidad y capacidad de cambio que evalúa un correcto diseño de la aplicación y la confiabilidad y capacidad de prueba que evalúa ciertas características de codificación.

CAPITULO 3: IDENTIFICACIÓN Y MEDICIÓN DE LA DEUDA TÉCNICA

En el presente capítulo se realiza una descripción de las herramientas que se han considerado para validar el método SQALE, para ello se seleccionan las más apropiadas para el contexto, ya que además de analizar la calidad de código y diseño del software, deben soportar el método SQALE.

3.1 Herramientas para el análisis de calidad de software.

Para elegir que herramientas se utilizan como prueba para la identificación y medición de deuda técnica, se comparan todas las herramientas en relación con las métricas de cada una que se estudiaron en el capítulo 2, para ello se han elegido las siguientes métricas indicadas en la Tabla 12 para el desarrollo del presente trabajo, ya que representan los factores de salud que comúnmente se presentan en el desarrollo de una aplicación software.

Tabla 12. Evaluación de herramientas con respecto a sus métricas.

Nombre	**Código**				**Diseño**			**SQALE**
	Complejidad	**Documentación**	**Duplicaciones**	**Tamaño**	**Violaciones Arq**	**Acoplamiento**	**Dependencias**	
CAST AIP	X			X			X	
SonarQube	X	X	X	X		X		X
FindBugs	X							
Structure 101	X	X					X	
Understand	X			X		X		
Sonargraph	X		X	X		X	X	
Source meter	X	X	X			X		
NDepend	X	X				X		
Lattix	X					X	X	
SourceMonitor	X	X		X				
SQUORE	X			X				X
Kiuwan	X	X	X	X		X		
Visual Paradigm	X			X				
MIA-Quality	X	X	X			X		X

Elaboración: El autor

Las herramientas que mejor se ajustan a las métricas seleccionadas son **SonarQube**, **SQUORE, Kiuwan y Mia-Quality**, además que efectúan el cálculo de deuda técnica del software, cuentan con documentación técnica para su aplicación, las únicas desventajas son

que herramienta **Kiuwan** no soporta el método SQALE y **SQUORE** y **MIA-QUALITY** no permiten su uso gratuito.

Por lo tanto se propone el uso de las herramientas **SonarQube**, **Kiuwan** para el cálculo de la deuda técnica de manera automática, y las herramientas **PMD** y **LocMetrics** para calcular violaciones de código fuente para con ello hacer el cálculo de la deuda técnica de manera manual. A continuación se describen de manera detallada las herramientas a utilizar para el cálculo de la deuda técnica.

3.1.1 SonarQube.

El objetivo principal de la plataforma SonarQube es la gestión de la "deuda técnica". La herramienta abarca 7 ejes de calidad de código.

- Arquitectura y Diseño
- Comentarios
- Reglas de código
- Duplicados
- Pruebas unitarias
- Errores potenciales
- Complejidad

Los resultados que presenta tras un análisis estático se visualizan en la Figura 23.

Figura 23. Pantalla de control de SonarQube

Fuente:(J. L. Letouzey, 2012)

SonarQube soporta 20 diferentes lenguajes, incluyendo ABAP, Android, C/C++,C-Sharp (C#), COBOL, Flex, Groovy, Java, JavaScript, Objective-C, PHP, PL/I, PL/SQL, Python, RPG, Swift, VB.NET, Visual Basic 6, Web (HTML, JSP/JSF),XML y esto se consigue a través de plugins que dispone la herramienta. Además la plataforma tiene la habilidad de añadir tus propias reglas en esos lenguajes.

Por otra parte la plataforma ofrece un seguimiento en el desarrollo del mantenimiento de un programa, y además de ello se puede utilizar para ejecutar análisis a partes de un programa en particular.

En cuanto al cálculo de la deuda técnica, mediante el uso del widget “evidencias y deuda técnica” se puede calcular un estimado del tiempo de corrección de los daños del código analizado, como indica la Figura 24.

Figura 24.Resumen de la deuda técnica de un proyecto

Y para conocer cómo se puede empezar esta reparación, podemos apoyarnos de los indicadores mencionados en la sección 2.1.1.6, como la pirámide, en la que nos indica la priorización de corrección de cada característica. La Tabla 13 muestra las entradas y salidas que se consideran en el análisis de calidad de software.

Tabla 13.Datos de entrada y salida de SonarQube.

Entradas	**Salida**
Git y SVN	Archivo JSON que contiene todos los problemas encontrados durante el análisis
.class archivos Java, .dll en archivos C#	

Elaboración:El Autor

3.1.2 Kiuwan.

Es una solución basada en cloud que permite realizar un análisis de calidad de software mediante un análisis estático.

Todos los principales lenguajes de programación (Java, C / C ++, JavaScript, PHP, C #, .NET, VB, de Objective-C, Cobol, SQL, etc.) son compatibles. Kiuwan también proporciona soporte para marcos populares como Android e Hibernate con conjuntos de reglas especializadas.

Kiuwan utiliza un modelo de calidad conocido como CQM[20] (Clinical quality measures), que se utiliza para la evaluación de la calidad interna del software y la extracción de la analítica del producto. Sugiere cinco indicadores: el mantenimiento, la fiabilidad, la portabilidad, la eficiencia y la seguridad, que se correlacionan con las características de software y proporciona visualizaciones llamativas sobre los resultados de análisis de calidad, además la herramienta Kiuwan contiene tres widgets relacionados con el cálculo de la deuda técnica, que se visualizan en la Figura 25.

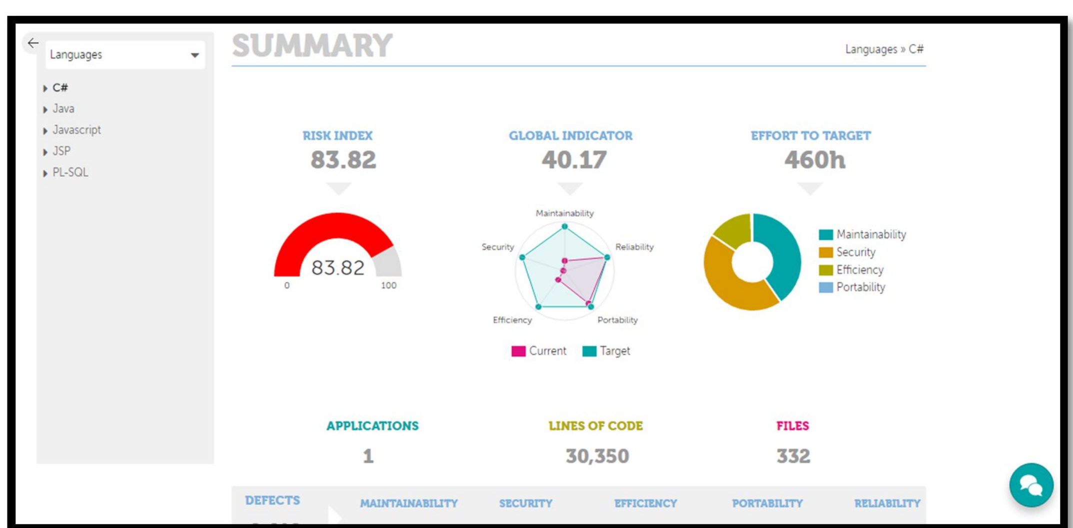


Figura 25.Widgets de Kiuwan sobre la deuda técnica.

Elaboración: El autor

El widget de la derecha indica el número de días necesarios para corregir todos los defectos encontrados para alcanzar su umbral de calidad indicador global, en este caso es 460 que es

[20] http://www.cqm-tech.com/

la suma del costo de cada característica de calidad. El indicador global del centro indica el grado de calidad del software, mientras mayor sea el número mayor calidad tiene el producto. Finalmente el indicador de la izquierda visualiza una combinación de ambos widgets de la derecha en el que informa el riesgo del proyecto. La Tabla 14 muestra las entradas y salidas que se consideran en el análisis de calidad de software.

Tabla 14. Datos de entrada y salida de Kiuwan

Entradas	Salida
.class archivos Java, .dll en archivos C#	Reportes .pdf, .csv

Elaboración: El autor

3.1.3 PMD.

La herramienta PMD es un producto de análisis estático que se utiliza para identificar posibles amenazas que son comunes como: las variables utilizadas, bloques catch vacíos, la creación de objetos innecesarios, y así sucesivamente. Es compatible con los lenguajes Java, JavaScript y en la Figura 26 se muestra un ejemplo de reporte tras realizar un análisis estático; en el que se indica el número de violaciones y la ubicación de las violaciones detectadas.

Summary

Rule name	Number of violations
ConfusingTernary	11
SingularField	1
UncommentedEmptyConstructor	11

Detail

PMD report

Problems found

#	File	Line	Problem
1	java\rest\controladores\AbstractFacade.java	19	Perhaps 'entityClass' could be replaced by a local variable.
2	java\rest\modelo\AproEstado.java	36	Document empty constructor
3	java\rest\modelo\AproEstado.java	71	Avoid if (x != y) ..; else ..;
4	java\rest\modelo\Aprobacion.java	56	Document empty constructor
5	java\rest\modelo\Aprobacion.java	130	Avoid if (x != y) ..; else ..;
6	java\rest\modelo\Departamento.java	44	Document empty constructor
7	java\rest\modelo\Departamento.java	96	Avoid if (x != y) ..; else ..;

Figura 26.Ejemplo de reporte generado por la herramienta PMD.

Elaboración: El autor

Esta herramienta es de licencia libre, además incluye un conjunto de reglas incorporadas y apoya la capacidad de escribir reglas personalizadas. Por lo general, los problemas comunicados por el PMD no son errores, sino código ineficiente, es decir, la aplicación puede funcionar correctamente, incluso si no se corrigieron. La Tabla 15 muestra las entradas y salidas que se consideran en el análisis de calidad de software.

Tabla 15. Datos de entrada y salida de PMD

Entradas	Salida
Dirección de código fuente	Reportes .xlm, html
.class archivos Java,	

Elaboración: El Autor

3.1.4 LocMetrics.

Esta herramienta es gratuita y permite hacer análisis al código de un proyecto de software para averiguar la cantidad de líneas de código que posee el software, además identifica algunos defectos que pueda contener el proyecto, como la complejidad, tal como se indica en la Figura 27.

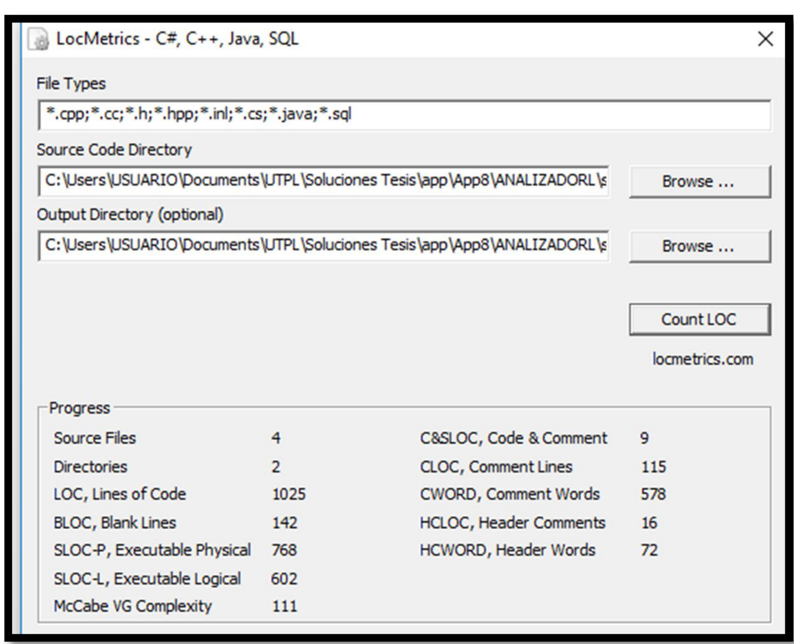


Figura 27. Ejemplo de reporte generado por la herramienta LocMetrics

Elaboración: El autor

Para realizar el siguiente análisis se utiliza esta herramienta para calcular el tamaño de las aplicaciones y los archivos con los que cuenta, es decir las clases, para proceder a realizar

los cálculos manuales de la deuda técnica. La Tabla 16 muestra las entradas y salidas que se consideran en el análisis de calidad de software.

Tabla 16 Entrada y salida de datos de la herramienta LocMetrics

Entradas	Salida
Dirección de código fuente	Reportes .xls, html
.class archivos Java,	

Elaboración: El autor

3.2 Análisis estático de software a través de herramientas de calidad de software.

Para validar las herramientas seleccionadas en el presente Capítulo se procede a analizar y evaluar a las mismas con proyectos desarrollados por estudiantes de la titulación de sistemas UTPL, con el fin de controlar la calidad de los mismos.

3.2.1 Aplicaciones de software a evaluar.

Los proyectos seleccionados para el presente análisis están codificados en lenguajes de programación, tales como: Java, PHP y Phyton. A continuación se detallan cada una de ellos asociándolos un id y ubicando las características de desarrollo

Tabla 17.Soluciones de software para el análisis de calidad.

Aplicación	Lenguaje	Características	Descripción	Procedencia
App1	**Java**	- **Estilo arquitectónico:** REST - **Patrón de arquitectura:** Facade	Aplicación de calificación.	**UTPL**
App2	**Java**	- **Patrón de arquitectura:** MVC	Prácticas de programación avanzada 4to ciclo	**UTPL**
App3	**PHP**	- **Framework:** CodeIgniter	Sistema Movilizacion	**UTPL**

		- **Patrón de arquitectura:** MVC		
App4	**Java**	- **Estilo arquitectónico:** REST	Buscador de servicios web	**Estudiante**
App5	**PHP**	- **Framework:** CodeIgniter - **Patrón de arquitectura:** MVC	Gestión de propuestas de tesis	**Estudiante**
App6	**Python**	- **Framework:** Django - **Patrón de arquitectura:** MVC	Visualización de datos abiertos sobre censos de discapacidades del ecuador.	**Estudiante**
App7	**Java**	- **Arquitectura**: 3 capas	Gestión de conductores de cooperativa de transporte	**Estudiante**
App7	**Java, JavaScript, HTML**	- **Estilo arquitectónico:** REST - **Patrón de arquitectura:** Facade	Buscador de servicios web	**Estudiante**
App8	**Java**		Analizador léxico y sintáctico.	**Estudiante**

Elaboración: El autor

Las aplicaciones descritas en la Tabla 17 son sometidas a un análisis estático con varias herramientas de análisis de calidad de software, seleccionadas en el presente capitulo con el objetivo de analizar los distintos resultados que presentan cada una de ellas.

3.2.2 Configuración para uso de herramientas.

Seguidamente se detalla el procedimiento que se utiliza para manejar las herramientas de análisis de calidad de software en el S.O. Windows.

- **SonarQube:** Esta herramienta es de uso gratuito, pero los plugins son de uso comercial, para poder realizar un análisis de software hacemos lo siguiente:
    - Descargarla versión 4.1.5 de "sonarqube" y el analizador "sonar-runner" de la página oficial http://www.sonarqube.org/downloads/ y extraerlos en C://.
    - Ejecutar el archivo StarSonar.bat ubicado en la dirección C:\sonarqube\bin\windows-x86-64
    - Dentro de la aplicación a analizar se crea un archivo con el nombre sonar-project.properties en el que se detallan los parámetros de análisis del proyecto http://docs.sonarqube.org/display/SONAR/Analysis+Parameters
    - Abrir el terminal y ubicarse dentro de la aplicación a evaluar, luego en la misma dirección se ejecuta el siguiente comando C:\sonar-scanner\bin\sonar-scanner.bat
    - Buscar los resultados dentro de un navegador en la siguiente dirección http://localhost:9000/

- **Kiuwan:** Esta herramienta permite un uso gratuito de las características básicas de análisis por un periodo de 15 días.
    - Para utilizar esta herramienta es necesario crear una cuenta https://www.kiuwan.com/signup/free/
    - Una vez dentro de la página subimos el proyecto, pero antes es necesario comprimirlo con la extensión .zip. Es posible analizar el proyecto de manera local o directamente desde la página, se elige la opción "Analize in the cloud", ya que los resultados de ambas formas se los visualiza en el dashboard de la página principal.

- **PDM:** El proceso de análisis de calidad mediante esta herramienta es el siguiente:
    - Descargarlos archivos de la herramientahttps://sourceforge.net/projects/pmd/files/pmd/4.2.6/pmd-bin-4.2.6.zip/download.
    - Se crea un archivo .xml en donde se agregan las reglas que se van a utilizar en el análisis.
    - Desde el terminal de Windows se escribe el siguiente comando pmd -d "C:\Users\USUARIO\Documents\UTPL\Soluciones Tesis\app\App1\PFT-WS-RESTful\src" -f summaryhtml -language java -R C:\Development\pmd-bin-5.5.1\MyRules.xml -r app1.html –shortnames, en donde, -d indica la dirección de los archivos del proyecto, el lenguaje se lo define con –languaje, y con –R

indicamos las reglas que se van a utilizar, en este caso indicamos la ubicación del archivo .xml que contiene las reglas de análisis de calidad de código y diseño.
    - Se obtiene un reporte de los resultados.

- **LocMetrics:** El proceso de análisis de calidad mediante esta herramienta es el siguiente:
    - Descargar la herramienta de la página oficial http://www.locmetrics.com/
    - Seguidamente al iniciar la herramienta se pide elegir el archivo o dirección del código fuente a analizar.
    - Finalmente se procede a realizar el análisis y a presentar un informa de los resultados obtenidos.

3.2.3 Análisis e interpretación de resultados de medición de la Deuda Técnica.

Para llevar a cabo este trabajo se efectúan las dos primeras fases de la gestión de la deuda técnica mencionadas en la sección 1.2.1, para las aplicaciones de la titulación de sistemas (no comprometidas con la universidad), ya que, el monitoreo es una tarea que requiere de tiempo y del conocimiento del código de las aplicaciones a evaluar, es por ello que aplican las tres fases de la gestión de la deuda técnica para las para las aplicaciones propuestas por el estudiante, y el monitoreo se lo realiza a través de dos iteraciones, para conocer la evolución de calidad que tendrán las aplicaciones por cada mejora que se aplique.

Siguiendo el proceso definido en la Sección 2.1, se describen a continuación los resultados del análisis de deuda técnica mediante cada actividad de la misma.

3.2.3.1 Fase de Identificación.

Se define la deuda técnica en unidad de medida días/hombre, ya que es una medida estándar que se puede adaptar a cualquier organización y dependiendo de eso se puede tener una estimación a nivel financiero de cuánto cuesta corregir un producto software. Se toma como un día a 8 horas de trabajo diario (ligado al número de horas de trabajo de nuestro medio).

Siguiendo el método SQALE en el modelo de calidad (ver Sección 2.1.1.3) se definen las métricas para evaluar las aplicaciones (ver Anexos *2*, Anexos *3*, Anexos *4*), según el lenguaje

de cada proyecto; y se las organiza en base a las características y sub-características que propone SQALE.

Para la evaluación con la herramienta SonarQube se utiliza la versión 4.1.2, ya que esta integra el método el SQALE, y las nuevas versiones utilizan un nuevo plugin llamado Governance[21] con licencia de pago. Las aplicaciones se evaluaron con el perfil de calidad SonarWay[22] que cuenta con 132 reglas de programación. En la herramienta Kiuwan se utiliza las reglas por defecto de la herramienta, que corresponden a las características de calidad mantenimiento y confiabilidad. Y en la herramienta PMD, se escogieron las reglas[23] (ver Anexo C) que se van a analizar, mediante el modelo de métricas indicadas en la Tabla 10, en la que se clasifico las reglas en base a la característica y subcaracteristica correspondiente. Los costos de remediación y no remediación se asignaron en base modelo de análisis de la Sección 2.1.1.4.

Como entrada de datos se toman los archivos de código fuente de las aplicaciones, de la siguiente manera:

- Java: Todas la aplicaciones Java están desarrolladas en NetBeans, para lo cual se evalúan los archivos ubicados en /src/…", las aplicaciones.

- PHP: Estas aplicaciones están desarrolladas mediante el framework CodeIgniter que sigue una arquitectura MVC, por lo tanto se evalúan los controladores, los archivos del modelo de datos y las vistas.

- Python: Esta aplicación se desarrolló mediante el framework Django y se evalúan los archivos ".py".

Posteriormente se realiza el proceso de identificación de DT a nivel de código y diseño en un mismo análisis mediante las herramientas seleccionadas, debido a que las reglas de programación que manejan contienen reglas tanto de código como de diseño. Los resultados se muestran en los Anexos ***5***, Anexos 6 y Anexos ***7***; las violaciones encontradas se las clasifica según la severidad y el número de repeticiones por cada una. En los Anexos 8 se indican las reglas de programación que más se han quebrantado en todas las aplicaciones, en la cual se muestra que unos de los mayores problemas al desarrollarlas son: la falta de

[21]https://www.sonarsource.com/why-us/products/plugins/governance.html
[22]http://docs.sonarqube.org/display/SONAR/Quality+Profiles
[23]http://pmd.sourceforge.net/snapshot/pmd-java/rules/

comentarios, duplicaciones, excesivo tamaño de métodos, números mágicos sin declarar, clases muy dependientes y clases fuertemente acopladas. Una vez detectada la deuda técnica en las aplicaciones se procede a medir el costo de pago de la misma, por lo cual dicho proceso de detalla en la fase de medición.

3.2.3.2 Fase de Medición.

Las cifras obtenidas por cada herramienta de análisis estático seleccionada se muestran a continuación, en donde se indica la herramienta que se ha utilizado, las líneas de código de cada proyecto, el número de archivos que maneja, el número de violaciones encontradas, y el desglose de la deuda técnica por cada característica, la suma de las características seleccionadas y la suma total de todas las características que calcula la herramienta.

Tabla 18.Resultado de análisis de deuda técnica por herramienta.

Herramienta	IDApp	LOC	Files	Num. evidencias	Deuda técnica por características				DT total
					Mantenibilidad	Confiabilidad	Capacidad de cambio	Capacidad de pruebas	
Versión 1									
SonarQube	App1	1572	18	1	0	20min	0	0	20min
	App2	2200	12	446	1d 2h	5h 50min	3d 3h	10min	5d 3h
	App3	4076	90	418	1d 4h	2d 7h	6h	51min	5d 2h
	App4	200	4	3	25min	20min	0	0	45min
	App5	1104	38	146	4h 2min	1d 2h	0	0	1d 6h
	App6	670	16	21	1h 56min	0	30min	11min	2h 37min
	App7	1393	17	276	2d 4h	3h 25min	2d 1h	0	5d 1h
	App8	768	4	252	2d 6h	1d	28d	30min	32d
Kiuwan	App1	1572	20	160	135h	0	26h 21min	0	162h
	App2	2200	12	391	422h	0	29h 21min	0	451h
	App3	8936	90	711	570h	0	25h 48min	0	595h
	App4	200	4	34	22h 24min	0	3h 06 min	0	25h 30min

	App5	1574	38	471	481h	0	11h 15min	0	492h
	App6	0	0	0	0	0	0	0	0
	App7	1393	17	270	256h	0	47h26 min	0	304h
	App8	768	4	170	137h	0	29h 12min	0	167h
PMD, LocMetrics	App1	1572	18	361	28 h	0	2h	0	30 h
	App2	2200	12	476	33h 25min	1h 8min	2h 1min	0	37 h 2 min
	App4	200	4	40	3h	0	40min	0	3h 40min
	App7	1393	17	340	23h	3h	3h 10min	0	28h
	App8	768	4	117	6h	2h 17min	2h 38min	0	10h 55min
Versión 2									
SonarQube	App4	200	4	2	20min	5min	0	0	25min
	App5	878	38	89	2h 25min	4h 33min	0	0	6h 58min
	App6	670	16	11	16min	0	30min	11min	57min
	App7	1386	17	225	3h 59min	3h 25min	2d 1h	0	3d
	App8	518	4	116	3h 41min	4h 5min	1d	20min	2d
Kiuwan	App4	211	4	22	17h 48min	0	2h 05 min	0	19h 54min
	App5	717	38	471	45h 54min	0	2h 51min	0	48h 45min
	App6	0	0	0	0	0	0	0	0
	App7	1387	17	223	149h	0	35h 03min	0	184h
	App8	545	4	147	104h	0	30h	0	134h
PMD, LocMetrics	App4	200	4	5	8min	0	42min	0	50min
	App7	1393	17	199	12h	58min	2h 77min	0	1d 9h
	App8	520	4	100	6h	50min	2h 38min	0	1d 1h

Elaboración: El autor

La Tabla 18 presenta el resultado en dos versiones del análisis de la deuda técnica de todas las aplicaciones, expuesto por cada herramienta, la primera versión es de todas las herramientas y la segunda es de las aplicaciones desarrolladas por el estudiante, en donde

se realizan mejoras de código para disminuir la deuda técnica. La corrección de las aplicaciones se la realizo mediante los siguientes criterios:

- La severidad de la regla debe ser mayor o menor.
- El valor del costo de remediación total por regla debe ser mayor a 50min.
- Las violaciones no deben pertenecer a falsos positivos.

En base a los criterios señalados y la herramienta de análisis de calidad de software, las reglas que se valoran para la corrección de las aplicaciones y la disminución de la deuda técnica se indican en el Anexos 8.

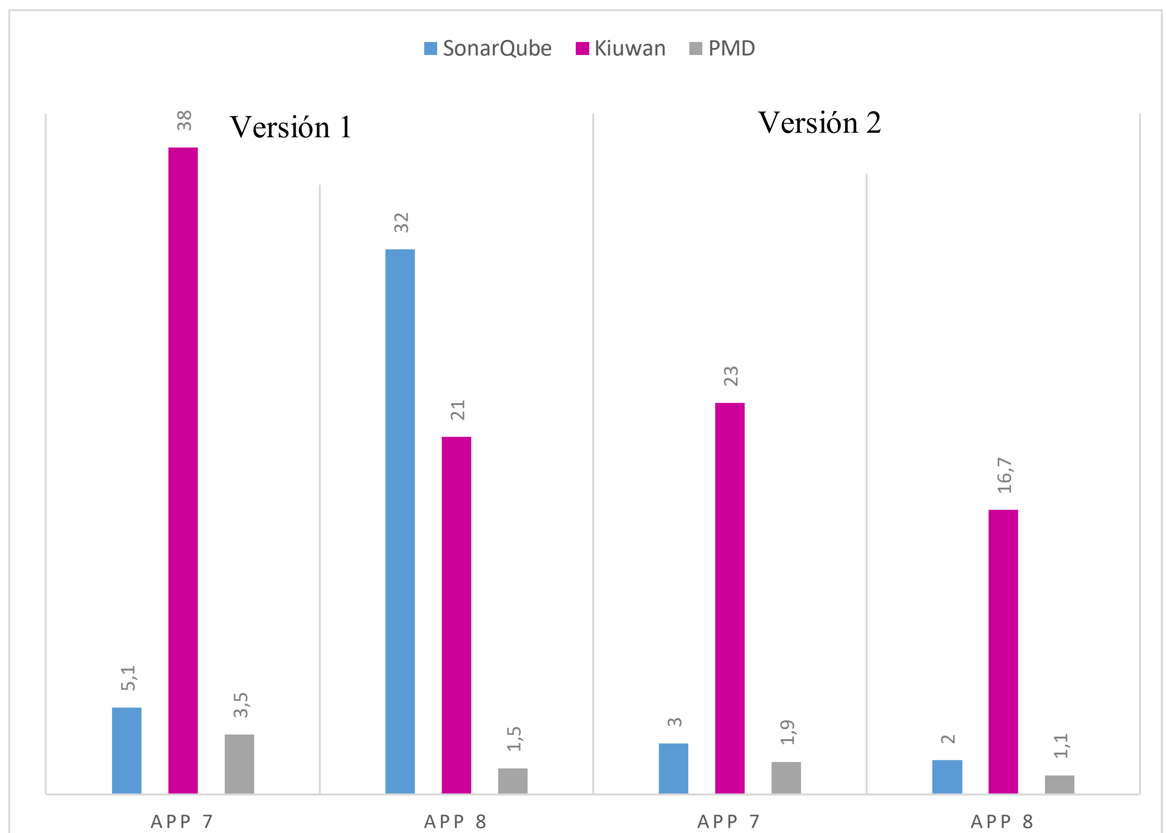


Figura 28. Deuda técnica por herramienta de las aplicaciones 7 y 8

Elaboración: El autor

La Figura 28 visualiza los cambios que se dan tras la evolución de las aplicaciones 7 y 8, que son las que presentan los resultados más significativos de deuda técnica, y se identifica que el valor de deuda ha disminuido considerablemente tras los cambios realizados, aunque los valores por herramienta varían mucho, lo cual se debe al número de reglas que emplea cada una por característica.

Mediante el valor de deuda técnica de las características de calidad calculadas se observa que actividad de ciclo de vida se encuentra con mayor deficiencia. La Figura 29 muestra el promedio de deuda técnica característica de todas las aplicaciones.

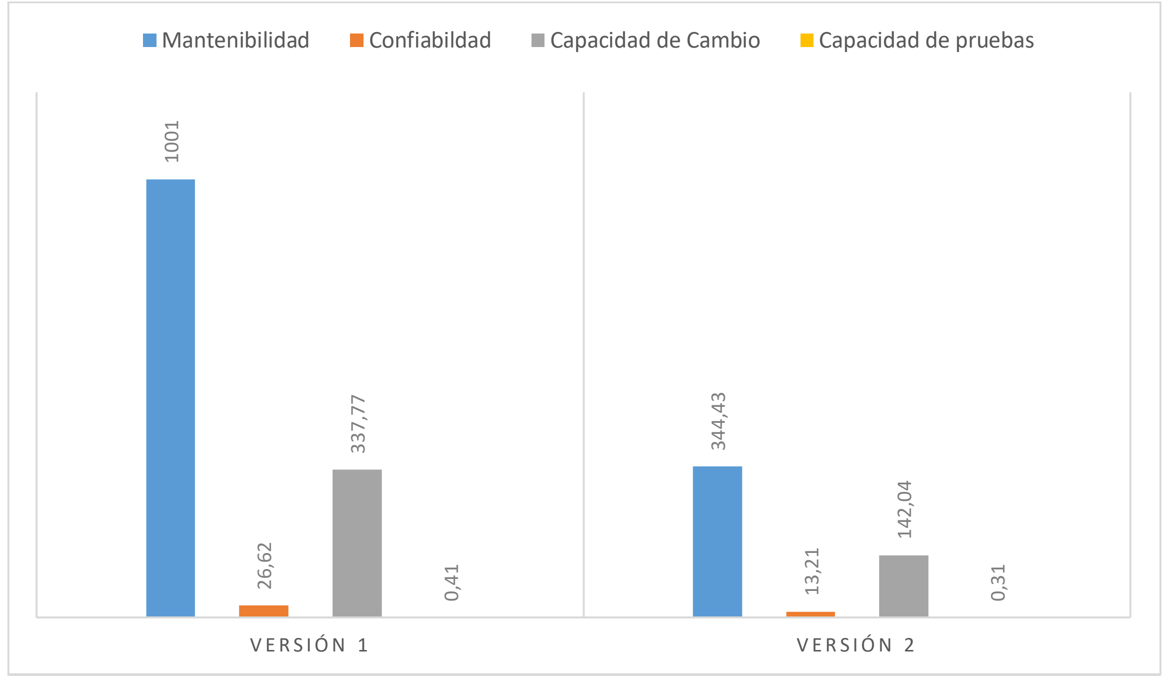


Figura 29. Promedio de deuda técnica por característica de todas las aplicaciones.

Elaboración: El autor

El resultado de las cuatro características mantenibilidad, confiabilidad, capacidad de cambio y capacidad de pruebas, la de mayor costo de remediación es la capacidad de cambio, lo que indica que las aplicaciones presentan problemas de evolución, es decir, no podrán responder de manera ágil a los cambios que requieran las aplicaciones frente a cualquier cambio en las mismas.

El valor de cada característica de calidad se obtiene mediante la suma de las violaciones que correspondan a la misma, tomando los datos de Anexos 5 se calcula el total de DT por característica, por ejemplo para la App 8 (Analizador Léxico), para lo cual se aplica la (Ecuación 1).

$$\mathbf{SCI} = (2\ x\ 10) + (213\ x\ 60) + (1\ x\ 10) + (118\ x\ 5) + (6\ x\ 10) = 12780\ min$$

$$\mathbf{SMI} = 19\ x\ 10\ +\ 10\ x\ 60\ +\ 10\ x\ 20\ +\ 4\ x\ 5\ +\ 2\ x\ 5\ +\ 2\ x\ 1\ +\ 1\ x\ 5\ +\ 1\ x\ 30\ +\ 1\ x\ 30 \\ +\ 1\ x\ 2\ +\ 1\ x\ 20\ +\ 11\ x\ 1\ +\ 8\ x\ 5\ +\ 6\ x\ 10\ +\ 2\ x\ 1\ +\ 1\ x\ 2\ +\ 1\ x\ 10 \\ +\ 3\ x\ 20 = 800\ min$$

$\mathbf{SRI} = 5\,x\,10 + 2\,x\,20 + 2\,x\,30 + 1\,x\,5 + 1\,x\,5 + 1\,x\,5 + 1\,x\,30 + 20\,x\,10 + 3\,x\,5 = 410\,min$

$\mathbf{STI} = (3\,x\,10) = 30\,min$

Luego se realiza el cálculo del total la deuda técnica aplicando la (Ecuación 2):

$$\boldsymbol{SQL} = SCI + SMI + SRI + STI$$
$$\boldsymbol{SQL} = 12780 + 800 + 410 + 30 = 15194\,min$$

Como el resultado esta expresado en minutos se convierte a horas dividiendo el total para sesenta, ya que una hora equivale a 60 minutos dando un total de 253,23 horas, pero como se ha definido un día de trabajo como 8 horas se divide el total en horas para 8, dando un resultado de 31 días con 7 horas, que equivale a 32 días de trabajo, que es el total de deuda técnica de la App 8.

Como estrategia de negocio el método SQALE implementa los índices consolidados (ver Sección 2.1.1.5) que se forman a partir de los resultados de deuda técnica por cada característica, que permiten tomar decisiones para la corrección de errores de software.

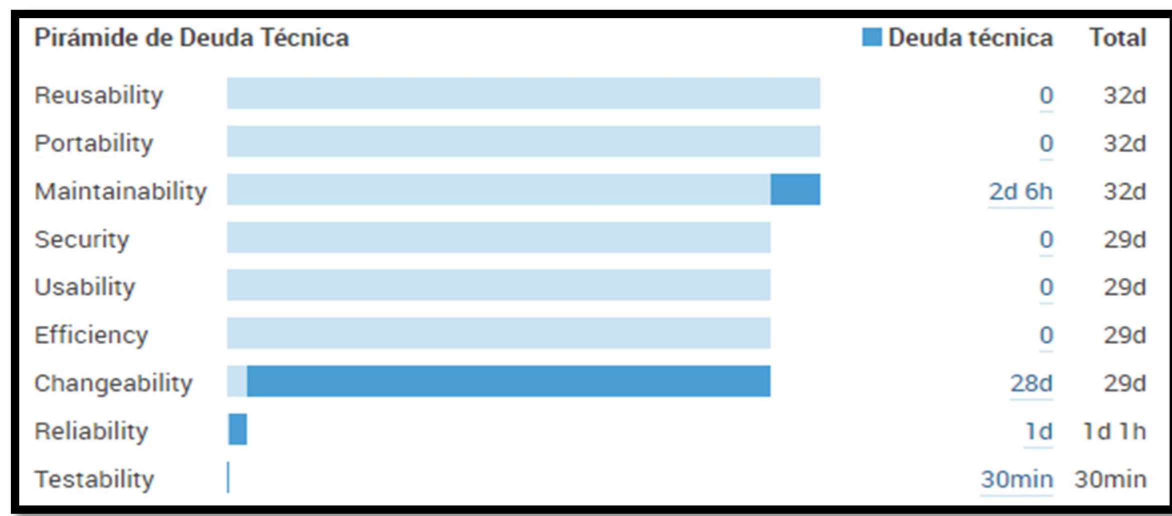


Figura 30.Pirámide SQALE del primer análisis de DT de aplicación Analizador Léxico.

Elaboración: El autor

Para calcular los indicies consolidados de la App 8, se aplican las Ecuaciones (Ecuación 3)(Ecuación 4),(Ecuación *5*)(Ecuación *6*), expuestas en la Tabla 8.

SCTI = STI = 30 min

SCRI = STI + SRI =30 + 410 = 440 min

SCMI = STI + SMI + SCI= 30+410+800 = 1240min

SCCI = STI + SRI + SCI =30+410+12780 = 13220min

Como se muestra en la Figura 30 al igual que el valor de los índices consolidados de la App 8, se requiere un mayor esfuerzo en el la capacidad de cambio del producto, lo que indica que esta aplicación puede tener problemas de evolución, es decir, se requiere de un mayor esfuerzo al aplicar cualquier cambio o mejora del producto.

Las reglas para los lenguajes Java y PHP que tienen influencia en las aplicaciones se observan en la Figura 31. Se puede visualizar que las aplicaciones se ven afectadas en mayor grado por las duplicaciones, que afecta al diseño y los comentarios obligatorios aumentando el esfuerzo de mantenimiento.

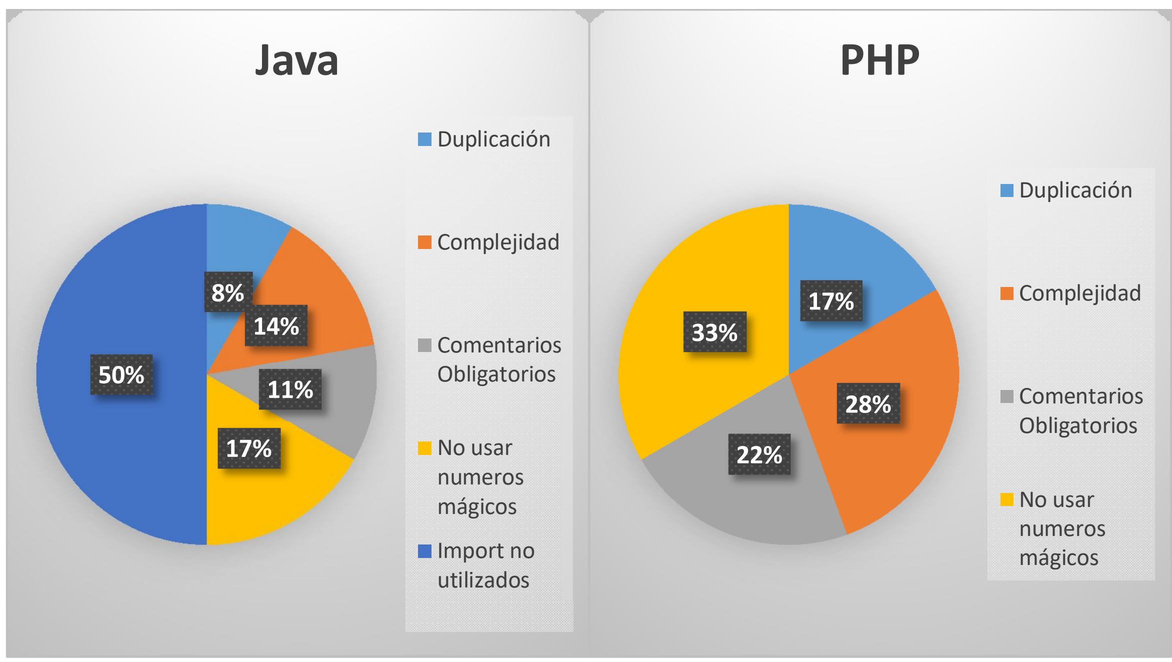


Figura 31. Métricas con mayor influencia en aplicaciones.

Elaboración: El autor

3.2.3.3 Fase de Monitoreo.

Para realizar esta fase se efectúa nuevamente el análisis de DT solo para las aplicaciones corregidas, que son desarrolladas por el estudiante. Y a continuación se describen los factores

de código y diseño que intervienen en el desarrollo y evolución del software, para observar los cambios obtenidos en las distintas versiones.

- **Tamaño**

La Figura 32 muestra la variación de tamaño de LOC en las aplicaciones 6 y 8, que son aquellas que presentan mayores cambios en el tamaño del software, debido a que en sus primeras versiones contenían un elevado valor de código duplicado.

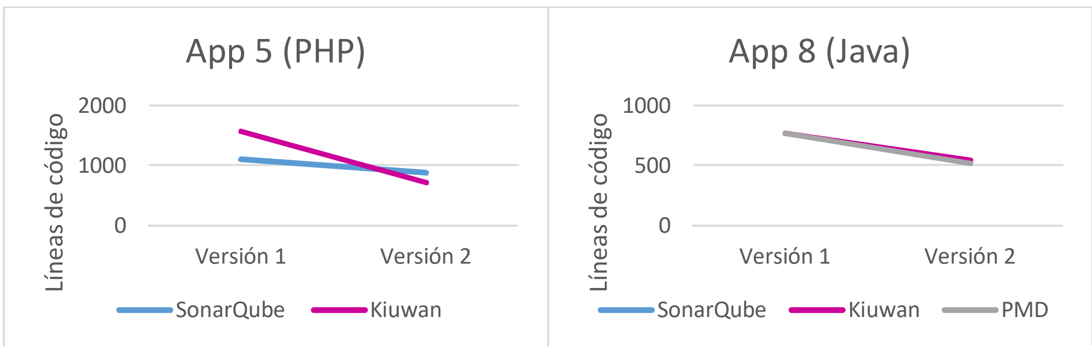


Figura 32. Variación de tamaño de aplicaciones 6 y 8 tras aplicar cambios para disminuir la deuda técnica.

Elaboración: El autor

Los datos permiten observar que el tamaño de las aplicaciones disminuye al aplicar iteraciones de cambios para reducir la deuda técnica, lo cual indica que la optimización de los programas tiene relación con la reducción y eliminación de partes de código innecesario, además esto contribuye a mejorar la capacidad del mantenimiento, ya que, mientras mayor es el tamaño de la aplicación, incrementa es esfuerzo necesario para corregirlo.

La variación de LOC por herramienta se debe al número de violaciones presentadas por cada una. La Figura 33 muestra el número de violaciones encontradas por herramienta.

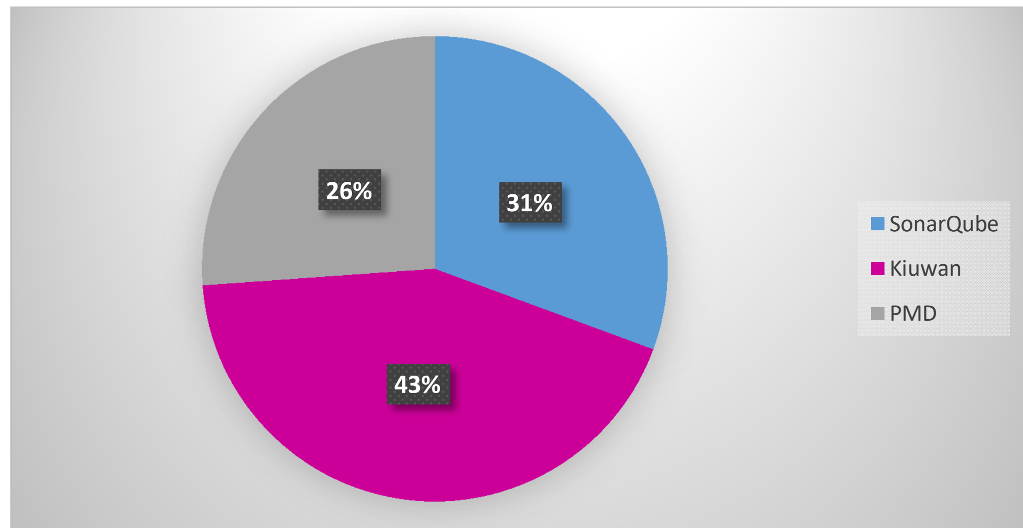


Figura 33. Número de violaciones por herramienta.

Elaboración: El autor

En cuanto a la cantidad de clases utilizadas, no existe ninguna variación, debido a que en la corrección de las aplicaciones no fue necesario cambios en la estructura.

- **Comentarios**

Al escribir código generalmente se suelen realizar comentarios explicativos del mismo, pero no siempre de manera correcta. En la Figura 34 se muestra el número de violaciones encontradas sobre comendatarios requeridos. Este presenta un problema que afecta en la mantenibilidad y en la capacidad de cambio de los programas, y aunque no representa un gran tiempo de remediación, el número de violaciones de esta regla hace que incremente el valor de duda técnica.

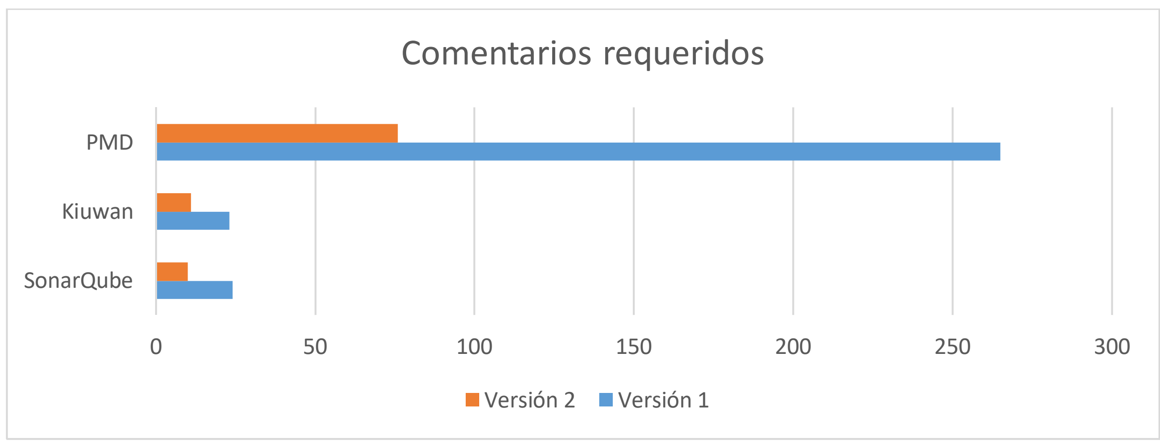


Figura 34. Variación de número de violaciones de comentarios requeridos.

Elaboración: El autor

Las herramientas SonarQube y Kiuwan muestran un menor número de violaciones de la métrica “comentarios requeridos” y esto es porque sus reglas solo consideran comentar los métodos que se utilizan, sin embargo las reglas de PMD exigen comentar todas las variables, y muchas de ellas son variables generadas automáticamente por los IDE’s al generar interfaz gráfica.

- **Complejidad**

La complejidad en un sistema suele aumentar cuando evoluciona el mismo, pero mediante la identificación de la deuda técnica, se procura reducirla para mantener el sistema estable, la Figura 35 representa el número de métodos que contienen complejidad ciclomática.

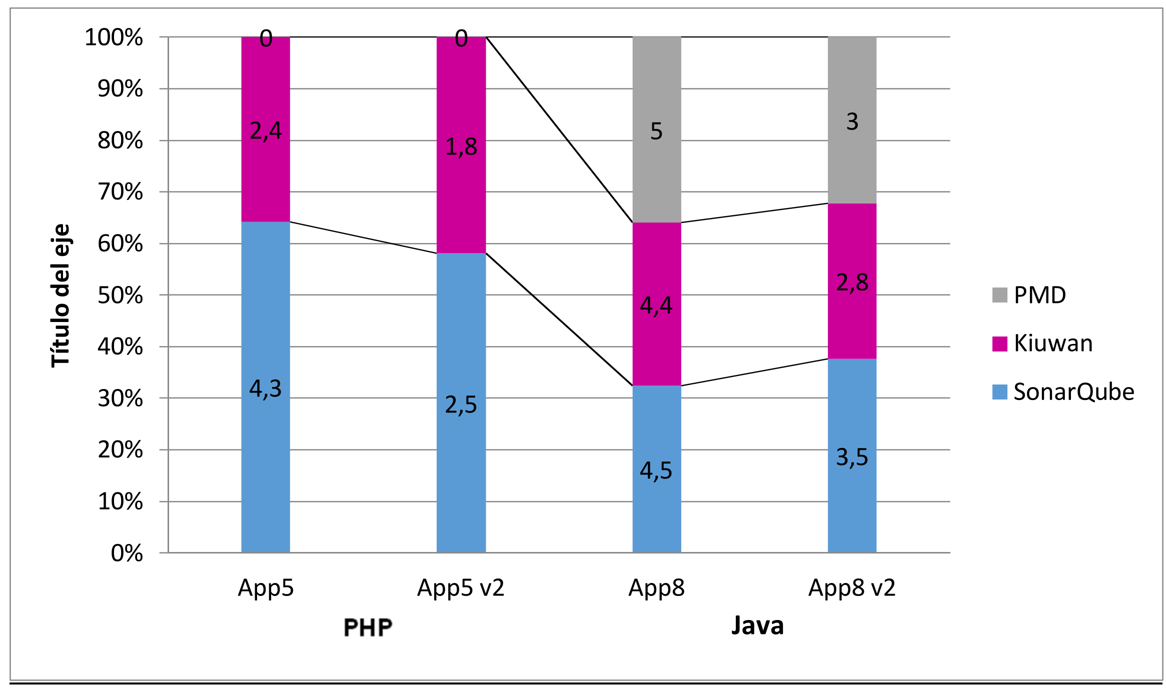


Figura 35. Complejidad ciclomática de las aplicaciones 5 y 8.

Elaboración: El autor

Para cada cambio de versión se ha logrado recudir la complejidad ciclomática en un 50%, puesto que los métodos desarrollados no pueden ser cambiados en su totalidad, ya que cambiaría su función original. El tamaño por cada método se reduce al reducir la complejidad, debido a que el incremento de blucles a más de incrementar la complejidad, incrementa el tamaño medio por método, que es evaluado por Kiuwan mediante la regla “Follow the limit for

number of statements in a method”[24], que permite un número máximo de 25 LOC por cada método.

- **Pruebas unitarias**

Antes de realizar un análisis de las características de calidad como la confiabilidad es necesario conocer si el software es verificar si el producto cumple con los requisitos de capacidad de prueba, ya que todas las características se sustentan en esta, lo que significa que si no es comprobable es porque es demasiado complejo o tiene muchas duplicaciones, etc. Y en el presente análisis no se encontraron valores significativos sobre esta característica, lo cual indica que la salud de las aplicaciones no presenta muchas deficiencias.

- **Acoplamiento**

Un factor que afecta al diseño de un sistema es el nivel de acoplamiento entre clases. La Figura 36 muestra el grado de acoplamiento entre clases de las aplicaciones 7 y 8, escritas en lenguaje Java, que contienen un mayor número de acoplamiento.

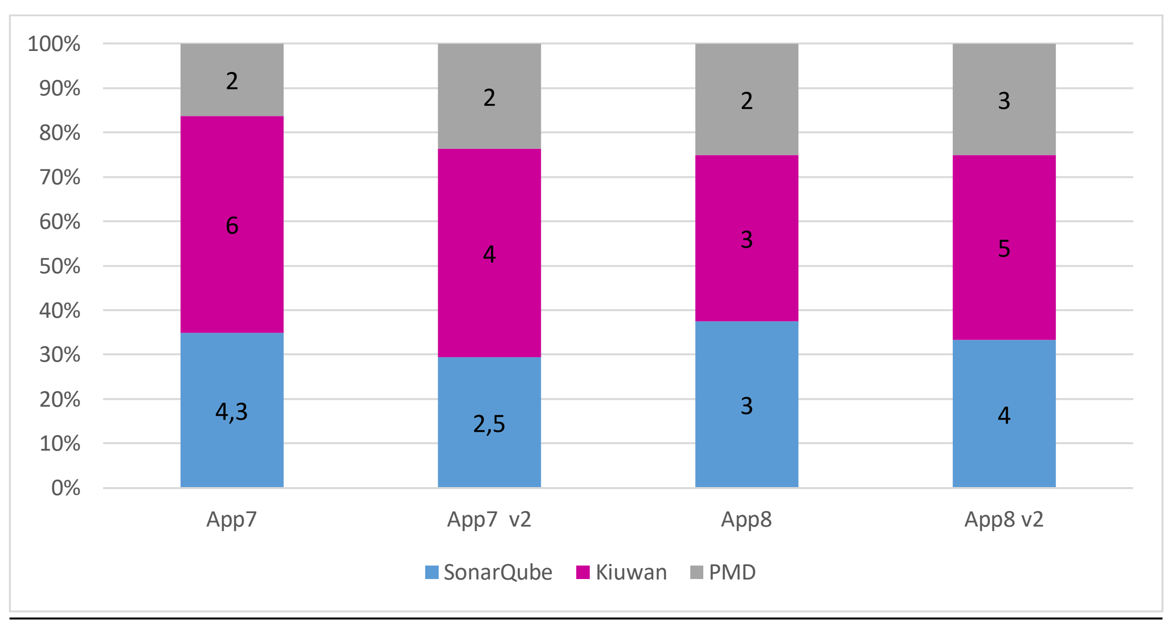


Figura 36. Número de clases con problemas de acoplamiento de las aplicaciones 7 y 8.

Elaboración: El autor

[24] Esta métrica perteneciente a la característica de mantenibilidad, manifiesta que debe existir un número máximo de 25 líneas de código por método para que no exista complejidad.

Las clases que tienen mayor acoplamiento son las de la App 7 que contiene 17 clases y en los resultados de Kiuwan se encuentra que 6 de ellas tienen un mayor grado de acoplamiento lo cual genera mayor dificultad en la reutilización de dichas clases y en el entendimiento aislado de cada una. Para la corrección de estos problemas se requiere de una mayor cantidad de esfuerzo, ya que son cambios que se deben realizar a clases enteras. Para la herramienta Kiuwan un problema de acoplamiento equivale a 4 horas de deuda técnica, y en caso de 6 clases el valor de deuda corresponde a 3 días de trabajo. En el caso de un sistema con mayor tamaño, el resultado sería muy excesivo y es posible que se requiera refactorizar todo un sistema o la mayor parte del mismo.

- **Duplicaciones**

La Figura 37 representa las clases que presentan duplicaciones por cada herramienta para la App 8. El número de líneas de código se reducen mientras que el número de clases es estable.

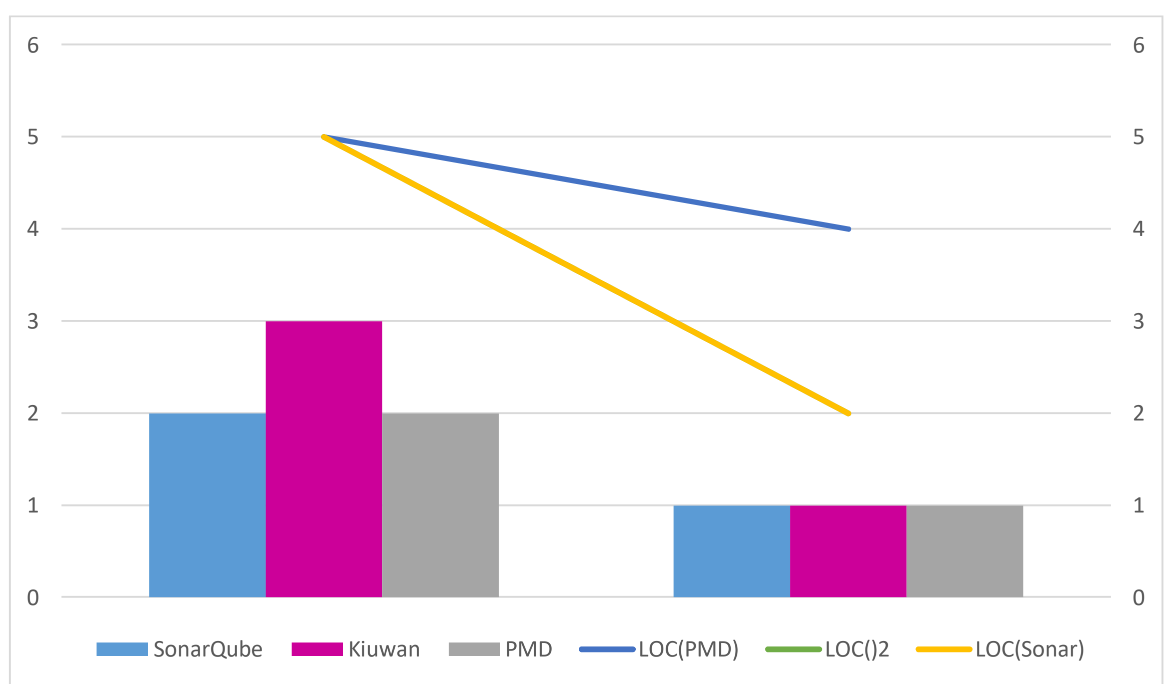


Figura 37. Resultado de duplicaciones de aplicación 8.

Elaboración: El autor

El código duplicado refleja un tipo de programación pobre, que afectan al mantenimiento de un software y los costos de mantenimiento son elevados, es por ello que al analizar las aplicaciones, es necesario revisar los resultados que se obtienen, debido a que algunas herramientas como PMD, detectan el código duplicado por medio de tokens, es por ello existen falsos positivos ya que, es probable que se detecten violaciones sin mucha relevancia,

como dos sentencias if o el manejo de excepciones que contienen partes similares de código. Mientras que con las demás herramientas la duplicación se maneja por bloques como se muestra en la Figura 38.

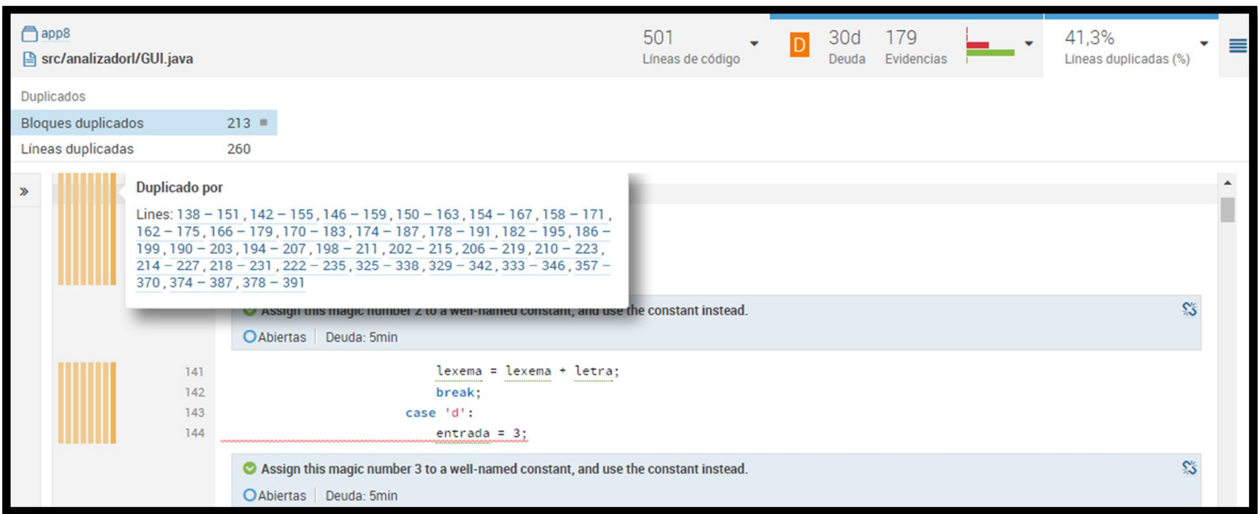


Figura 38.Resultados de código duplicado de Analizador Léxico con SonarQube.

Elaboración: El autor

El grado de defectos de un software se refleja mediante el ratio[25] de la deuda técnica, y de las herramientas utilizadas, la que proporciona tal calificación es SonarQube como se muestra en la Figura 39. Esta información es de gran valor, ya que mediante dicha calificación se puede categorizar las aplicaciones o módulos en caso de un sistema grande para la toma de decisiones, y para considerar un sistema como saludable, la calificación no debe descender del valor B, debido a que calificaciones mayores al valor C reflejan un valor de deuda técnica de muchos días, como la App 8, con calificación D, tiene una deuda de .32 días, sobrepasando un mes de trabajo.

[25] Ratio de la deuda técnica: Es el esfuerzo que se debería de invertir para reescribir todo el código desde el principio.

NOMBRE	RATIO DE DEUDA TÉCNICA	CALIFICACIÓN SQALE
app1	0.0%	A
app2	4.0%	A
app3	2.1%	A
app4	0.8%	A
app6	0.8%	A
app7	6.0%	A
app8	67.3%	D

Figura 39. Calificación SQALE de aplicaciones del primer análisis de DT.

Elaboración: El autor

El objetivo del método SQALE es gestionar adecuadamente la deuda técnica en un software, es por ello que es necesario utilizar las características que ofrece el método, para una correcta toma de decisiones, ya que es importante dar cumplimiento inmediato a las violaciones encontradas en un software y resulta benéfico el resolver la deuda técnica con el fin de tener un sistema adecuado y listo para cualquier cambio.

Finalmente tras visualizar los diferentes resultados obtenidos del análisis al código a través de las tres herramientas se puede concluir que los diversos resultados desde el punto de vista de cada herramienta (como las reglas de programación y los costos de remediación). Adicionalmente a continuación se recomiendan unas herramientas de uso no comercial que son de ayuda para calcular violaciones de código y diseño de un programa.

3.2.4 Otras herramientas.

Adicional a las herramientas utilizadas para el análisis, es necesario mencionar que existen otras herramientas de análisis estático que se utilizan para identificar los problemas de código, para realizar los cálculos de manera manual, tal como se lo hizo con las herramientas PMD y LocMetrics, entre las que constan por ejemplo

JArchitect proporciona una vista grafica de las dependencias entre los archivos, los números que aparecen en la matriz son número de miembros involucrados en el acoplamiento entre paquetes que se tiene que eliminar y los paquetes que están involucrados en los ciclos.

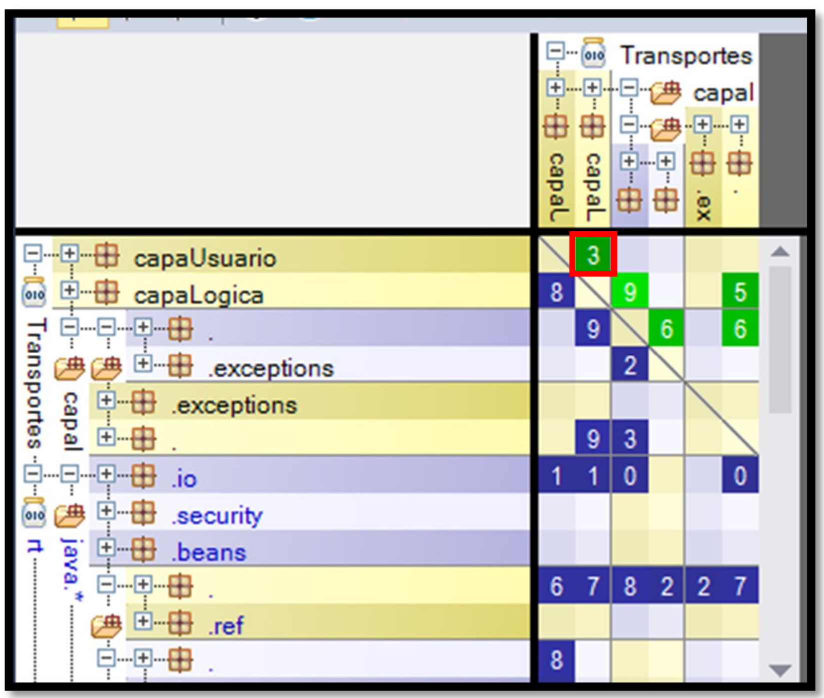


Figura 40. Análisis de diseño con Jarchitect para la App 7

Elaboración: El autor

La Figura 40 indica que en la App 7, entre la capa lógica y la capa de usuario se observa un peso de 3, el cual se genera por las dependencias que tienen la clase interfazGUI con las tres clases de la capa lógica.

Además la herramienta JArchitect proporciona una vista gráfica puntual sobre el código que actúa directamente con los archivos con dependencia, como se muestra en la Figura 39.

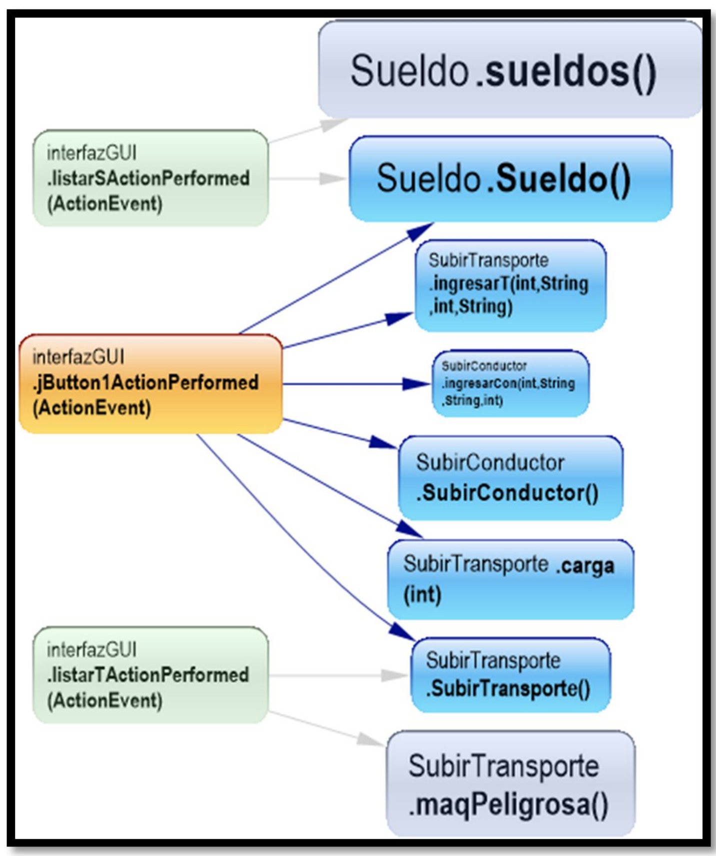
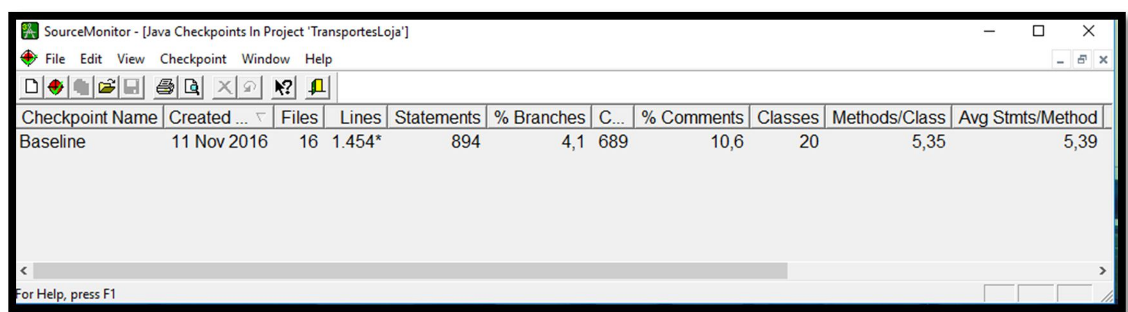


Figura 41. Dependencias de clases presentada por Jarchitect de la App7

Elaboración: El autor

SourceMonitor es un programa de uso gratuito que permite ver qué cantidad de código tiene una aplicación y también ver que complejidad tiene. La Figura 42 muestra el resultado de análisis de la App 7, en el que presenta las métricas principales que considera esta herramienta.



Checkpoint Name	Created ...	Files	Lines	Statements	% Branches	C...	% Comments	Classes	Methods/Class	Avg Stmts/Method
Baseline	11 Nov 2016	16	1.454*	894	4,1	689	10,6	20	5,35	5,39

Figura 42. Resultado de Análisis de la App 7 por medio de SourceMonitor

Elaboración: El autor

La Figura 43 muestra que los resultados que presenta SourceMonitor de manera gráfica.

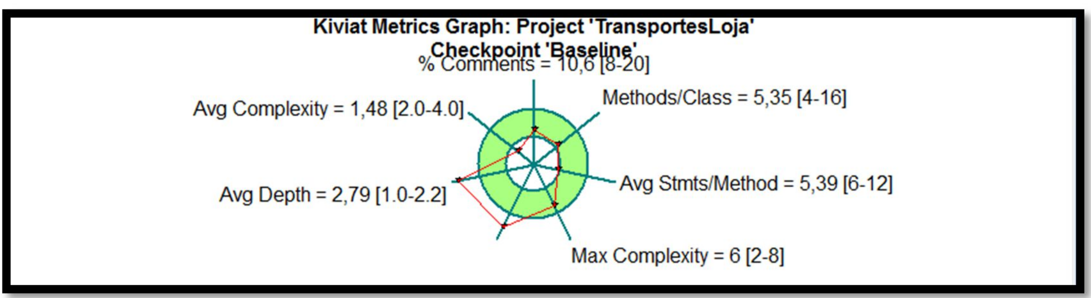


Figura 43. Resultado de forma gráfica de Análisis de la App 7 por medio de SourceMonitor

Elaboración: El autor

Estas herramientas no comerciales se pueden utilizar para realizar el cálculo de la deuda técnica, y adaptarlas a un proyecto de acuerdo a sus necesidades, pero para ello es necesario conocer el proceso que se debe llevar para realizar el cálculo adecuado y llevar un control del mismo, a través de las fases de la gestión de la deuda técnica. A continuación se presentan las conclusiones y recomendaciones que se han obtenido en el proceso de desarrollo del presente trabajo de investigación.

CONCLUSIONES

- El método SQALE se adapta a cualquier entorno donde se desee aplicar siempre y cuando el nivel de madurez del proyecto de software permita ir aumentando el nivel de características a evaluar.

- Las características de calidad a evaluar se deben escoger según el ciclo de vida del proyecto, ya que en este estudio se evalúa la "capacidad de prueba" pero no se encuentran resultados de errores dado que las aplicaciones no tienen Scripts de prueba, ni se consideró este atributo de calidad en ellas.

- El análisis estático utilizando como método SQALE es más directo, porque el software a evaluar no necesita ser ejecutado, basta con la evaluación del código fuente, aunque es necesario especificar los archivos que contienen el código desarrollado, ya que de existir código generado por IDE´s detecta errores en los mismos.

- De un total de 15 herramientas consultadas, el 71% de las herramientas encontradas son comerciales y tienen versiones de evaluación limitadas, con elevados costos de adquisición para la pequeña y mediana empresa. Así mismo el 29% de herramientas no comerciales, realizan análisis estático para identificar errores de código y de diseño, pero con el problema de que solo evalúan ciertas métricas de código.

- El valor de DT por cada herramienta difiere por el número de métricas que evalúa cada una, por esta razón así se analicen las mismas características de calidad, los resultados son distintos, además el valor y la prioridad que tiene cada regla viene dada por defecto, por ello los resultados de la herramienta PMD son considerablemente menores al resto de herramientas.

- El cálculo de la deuda técnica realizado manualmente es complejo y necesita una mayor cantidad de tiempo, porque en el proceso es necesario organizar las métricas a evaluar con las características de calidad del método SQALE y asignar costos a cada regla, mientras que por medio de las herramientas, es cuestión de esperar los resultados y validar que sean correctos.
- El porcentaje de deuda técnica obtenido mediante la sumatoria de todos los programas analizados (5) a través de las herramientas SonarQube y Kiuwan corresponde al 87%, mientras que la misma deuda pero con cálculo manual de todas la aplicaciones

evaluadas manualmente corresponde al 13%, se aprecia que existe una diferencia del 74%, esto se debe a la cantidad de reglas que utilizan por defecto las herramientas (243 reglas) a diferencia del cálculo manual donde se han utilizado 126, por esta razón es más factible utilizar una herramienta de análisis de calidad para obtener datos fiables y apoyarnos del cálculo manual para validar los mismos.

- Las métricas como la falta de comentarios en variables y métodos, la complejidad al tener un excesivo número de bucles en los métodos, un excesivo acoplamiento entre clases y una excesiva duplicación de código que se evaluaron en el presente trabajo permiten observar los errores que comúnmente se cometen en el desarrollo del software.

- Las aplicaciones analizadas en el presente trabajo se consideran pequeña debido a que el tamaño en líneas de código es menor a 2000, sin embargo, el uso de análisis estático mediante el método SQALE en un software con un nivel de madurez alto se puede llevar a cabo sin ningún inconveniente, dado que todo software complejo generalmente está construido por pequeñas partes de código.

A continuación, se concluye el resultado de la validación de cada herramienta:

- **SonarQube**

- La ventaja de utilizar la herramienta SonarQube es que se adapta a cualquier proyecto y lenguaje, además es una buena opción para adentrarse al conocimiento de la deuda técnica, es de fácil uso y presenta métricas que comúnmente se evalúan en un sistema y sus resultados se enfocan a la DT, así mismo lleva un control de versiones para comparar los cambios que se realizan en las mismas.

- La herramienta SonarQube permite la integración de reglas de programación u otros perfiles de calidad existentes como Sonarway, el cual se utilizó en el trabajo y cuenta con 120 reglas de programación para validar un sistema según las necesidades del proyecto.

- El análisis de un proyecto requiere la configuración de un archivo (sonar-project.properties) con las propiedades del mismo, y los proyectos desarrollados en NetBeans con varios módulos, deben estar organizados en diferentes directorios que contengan los recursos, caso contrario no es posible realizar el análisis, además si se

tiene un software con clases escritas en varios lenguajes de programación (por ejemplo Java y JS), se debe especificar en el archivo de configuración que el proyecto es multilenguaje.

- **Kiuwan**

- La herramienta Kiuwan permite en un solo análisis evaluar todo el código, independientemente del lenguaje y las reglas que evalúa es posible modificarlas, de acuerdo a la necesidad del proyecto.

- El cálculo de la deuda técnica en esta herramienta no utiliza el método SQALE, sino un modelo llamado CQM (Clinical quality measures), por ello que con esta herramienta solo es posible evaluar características de comprobabilidad y mantenibilidad, ya que el modelo además analiza la portabilidad, reusabilidad y seguridad, características que no se han considerado en este trabajo.

- Los resultados solo presentan el total de DT por violación, pero no indican cual es el valor mínimo con el cual se calcula, por lo tanto, esos datos se los debe adquirir manualmente.

- **PMD**

- Las desventajas de realizar el cálculo de la deuda técnica de manera manual, es que las herramientas gratuitas solo analizan ciertos tipos de lenguajes de programación, como PMD que solo analiza código fuente escrito en lenguaje Java, el proceso de selección de métricas a evaluar y el cálculo de DT, es complejo si el tamaño del software es demasiado extenso, el proceso puede tener una gran duración, lo cual provocará una desventaja de costos para las empresas de desarrollo.

- La diferencia de realizar el cálculo de DT manual y automático es que al momento de calcular los índices consolidados manualmente se presenta una leve diferencia en los resultados, con relación a los obtenidos con la herramienta SonarQube, debido a que esta última redondea los valores al convertir los resultados de minutos a horas.
- Al realizar el cálculo de las líneas de código, se deben definir los archivos específicos para su análisis, pues al evaluar todos los archivos del proyecto las herramientas

presentan resultados distintos, ya que herramientas como Kiuwan analizan otros archivos que no forman parte del código fuente.

- SonarQube, Kiuwan, PMD son útiles para identificar la DT dentro de proyectos software en el ámbito académico; en el ámbito laboral, una alternativa de herramienta open source es SonarQube, debido que el cálculo de DT llega a ser igual de fiable que el manual y porque proporciona indicadores que analizan el estado del proyecto, pero para ello se requiere de recursos humanos que se especialicen en el uso de la herramienta.

- SonarQube como herramienta permite configurar las métricas a evaluar y la asignación de costos para cada regla o violación; también permite la detección de falsos positivos, es decir, cuando la herramienta considera errores semánticos que son propios del lenguaje.

RECOMENDACIONES

De los resultados obtenidos tras la evaluación de la deuda técnica en aplicaciones de software se recomienda que:

- Para evaluar un proyecto y conocer cuál es la deuda técnica del mismo, es necesario identificar cuáles son las características que se adaptan al proyecto para evaluarlas, y con ello hacer una revisión de las métricas pertenecientes a cada característica a evaluar, ya que es probable que existan violaciones que no sean posible modificarlas e incrementan el valor de deuda, como por ejemplo hay código que se genera automáticamente por un IDE.

- El análisis de calidad de software se lo debe llevar a cabo por un equipo de control de calidad a través de revisiones y auditorías, en las cuales se analiza el estado del proyecto en base al valor de DT para sugerir a los programadores los cambios que se deben realizar para obtener una mejora del software.

- Los cambios se los hace en toda la aplicación y no solo en cierto fragmento de código, por lo que se recomienda dejar la configuración por defecto por cada herramienta, y si se desea realizar algún cambio, se debe considerar la experiencia del programador y el conocimiento que tenga del sistema.

- Antes de corregir las violaciones encontradas, hay que documentar el valor de deuda técnica de cada regla con el grado de severidad correspondiente, y si es necesario calcular los índices consolidados, ya que con estos datos es más fácil tomar decisiones para priorizar la corrección de los errores de código encontrados.

- Se recomienda llevar un control de versiones de la aplicación evaluada, ya que herramientas como SonarQube ofrece una comparación de versiones y así se puede observar que cambios se obtuvieron tras la mejora del código fuente o del diseño de un software.

ANEXOS

Anexos 1. Glosario de términos

Software: Programa o conjunto de programas que permiten ejecutar tareas en la computadora.

Aplicación: Es aquel que permite que el usuario realice tareas comúnmente realizadas por las personas.

Calidad de software: Es la cualidad o cualidades de un software para que determinan su correcta utilidad.

Sistema de software: Es el conjunto de programas o módulos que realizan algunas tareas.

Deuda Técnica: Es un concepto de programación que refleja el desarrollo pobre de un software.

Normalizar: Proceso de elaborar, aplicar y mejorar normas.

Modelos: Es un prototipo que sirve de referencia para crear un producto de la misma naturaleza

Método: Conjunto de estrategias que se utilizan para cumplir un objetivo.

Anexos 2. Reglas de código para el análisis de deuda técnica en lenguaje JAVA

Característica	Subcaracterística	Regla	Tipo de DT
Mantenibilidad	Comprensibilidad	Package declaration should match source file directory	Código
Mantenibilidad	Comprensibilidad	A field should not duplicate the name of its containing class	Código
Mantenibilidad	Comprensibilidad	Unused local variables should be removed	Código
Mantenibilidad	Comprensibilidad	Nested code blocks should not be used	Código
Confiabilidad	Logic	Conditions in related "if/else if" statements should not be duplicated	Código
Confiabilidad	Arquitectura	Methods should not be empty	Diseño
Confiabilidad	Arquitectura	The Object.finalize() method should never be overriden	Diseño
Confiabilidad	Arquitectura	Non-constructor methods should not have the same name as the enclosing class	Diseño
Confiabilidad	Arquitectura	Method parameters, caught exceptions and foreach variables should not be reassigned	Diseño
Confiabilidad	Arquitectura	"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method	Diseño
Capacidad de cambio	Arquitectura	Evitar un ciclo entre los paquetes de Java.	Diseño

Capacidad de cambio	Arquitectura	Campos variables de clase no deben tener acceso público	Diseño
Capacidad de cambio	Arquitectura	Las clases no deben ser acoplados a muchas otras clases (Responsabilidad Individual Principio)	Diseño
Capacidad de cambio	Arquitectura	El paquete sin nombre por defecto no se debe utilizar	Diseño
Capacidad de cambio	Arquitectura	Las clases abstractas sin campos deben ser convertidos a las interfaces	Diseño

Anexos 3. Reglas de código de la herramienta Kiuwan para el análisis de deuda técnica en lenguajes JAVA Y PHP

Caracteristica	Regla	Lenguaje	Severidad
Maintainability	Use accessor methods (getter/setter) for property access	Java	
Maintainability	Avoid many-to-many associations	Java	
Reliability	Avoid more than one entity mapped to same database table	Java	
Reliability	Avoid setting FlushMode that could produce stale data issues	Java	
Reliability	Every persistent class should be its own proxy	Java	
Reliability	Close sessions in the method where they are opened	Java	
Maintainability	Declare identifier property/properties on persistent classes	Java	
Reliability	Declare type for java.util.Date property in configuration file	Java	
Reliability	Implement a zero-argument constructor for persistent classes	Java	
Reliability	Map a Hibernate property type only to the corresponding Java type	Java	
Maintainability	Use one mapping file per persistent class	Java	
Reliability	Every class implementing a composite-* element should override equals() and hashCode()	Java	
Reliability	Classes referenced in the configuration file (*.hbm.xml) should be declared	Java	

Maintainability	Avoid domain entities using J2EE APIs	Java	
Reliability	Use interface for collection-valued properties	Java	
Reliability	You should always specify a name attribute for identifiers in Hibernate	Java	
Reliability	Use a nullable (non-primitive) type for identifier fields in persistent classes	Java	
Reliability	Use version instead of timestamp	Java	
Reliability	AdapterViews cannot have children in xml.	Java	
Reliability	Do not call a recycled resource.	Java	
Reliability	Be sure super method is called first on initialization methods	Java	
Reliability	Ensure that on finalization methods, super method is called at the end of the method.	Java	
Reliability	Transactions have to be completed calling commit () method.	Java	
Reliability	Avoid scrolling widgets with scrolling widgets children.	Java	
Reliability	Alias and resource types have to be the same.	Java	
Reliability	Avoid method calls using color id directly as paremeter.	Java	
Reliability	Cycles can't exist in resources definition.	Java	
Reliability	Check ScrollViews just have one children.	Java	
Reliability	Check if TextView is correctly used	Java	
Maintainability	Discourage use of Serialization, use JSON instead..	Java	

Reliability	Type check for views with an assigned id.	Java	
Reliability	Avoid assignments in while / do-while loop condition.	Java	
Reliability	Avoid loops without an initiator and an increase.	Java	
Maintainability	Provide Javadoc comments for public classes and interfaces.	Java	
Maintainability	Provide Javadoc comments for protected classes and interfaces.	Java	
Reliability	Call setRollbackOnly() if a method in a bean throws an application exception.	Java	
Reliability	Avoid concurrently iterations over a collection.	Java	
Maintainability	Avoid nesting more than three loops.	Java	
Maintainability	Avoid nesting more than 1 try / catch.	Java	
Maintainability	Only declare 'protected' constructors for abstract classes.	Java	
Reliability	Always test the type of object when it is converted to an abstract collection.	Java	
Reliability	Always check object type before cast.	Java	
Maintainability	Duplicated code: medium block	Java	
Reliability	Do not start, stop or manage objects in a EntityBeans java.lang.Thread.	Java	
Reliability	Do not use return statements inside catch blocks.	Java	
Reliability	Avoid empty final blocks.	Java	

Reliability	Avoid empty try blocks.	Java	
Reliability	Avoid java.lang.Error catch exceptions.	Java	
Reliability	Avoid capturing java.lang.Exception exceptions.	Java	
Reliability	Avoid capturing java.lang.IllegalMonitorStateException.	Java	
Reliability	Avoid capturing NullPointerExceptions.	Java	
Reliability	Avoid throwing java.lang.Error.	Java	
Reliability	Always make sure that the method 'next()' in an Iterator class can be cast java.util.NoSuchElementException.	Java	
Reliability	Avoid overloading the method finalize().	Java	
Maintainability	Avoid unnecessary finalize() method.	Java	
Reliability	Do not use instanceof to distinguish between exceptions.	Java	
Maintainability	Avoid multiple comparisons within if.	Java	
Maintainability	Too many uses of the negation operator (!) in a method.	Java	
Reliability	Avoid Exception, RuntimeException o Throwable in catch statements.	Java	
Reliability	Avoid synchronised blocks that have an invocation to notify() or notifyAll() as the statement.	Java	
Reliability	Avoid the synchronization of objects java.util.concurrent.locks.Lock.	Java	
Reliability	Avoid using this in synchronised blocks.	Java	

Reliability	Avoid using wait() method of the class java.lang.Object outside a while loop.	Java	
Reliability	Avoid creating new objects in static initializers before all static fields have been initialized.	Java	
Reliability	Use constructors than initialize all the fields in a class.	Java	
Reliability	Initialize all static fields.	Java	
Reliability	Close input and output resources in finally blocks.	Java	
Reliability	Avoid instantiating java.lang.ClassLoader.	Java	
Reliability	Always declare Junit.framework.TestCase.suite() as a public static method.	Java	
Reliability	Avoid problematic constructions in a while loop	Java	
Reliability	Avoid catch blocks with empty bodies.	Java	
Reliability	Avoid casting primitive data types to lower precision.	Java	
Maintainability	Avoid nested IF sentences with too many levels.	Java	
Reliability	Place default clause at the end of a switch - case.	Java	
Reliability	Avoid 'for' and 'while' sentences with empty bodies.	Java	
Reliability	Do not assign loop control variables in the body of a for loop.	Java	
Reliability	Avoid if statements with empty bodies.	Java	

Reliability	Provide 'default' label for each switch statement.	Java	
Reliability	Avoid using a switch structure with a bad case statement.	Java	
Maintainability	Avoid unconditional If blocks	Java	
Reliability	Return zero-length arrays instead of null.	Java	
Maintainability	Use a braches block for 'do-while' statements.	Java	
Maintainability	Use a braces block for 'for' statements.	Java	
Maintainability	Provide a branch block for 'if' statements.	Java	
Maintainability	Provide a braches block for 'while' statements.	Java	
Reliability	Avoid calling non-final, non-static and non-private methods from constructors.	Java	
Maintainability	Use only one condition statement in for blocks.	Java	
Reliability	Use 'for' loops with explicit condition and increment/decrement statements.	Java	
Reliability	Capture OutOfMemoryError for large arrays initializations.	Java	
Maintainability	Avoid if/else-if chains performing type testing.	Java	
Reliability	Avoid calling this.getClass.getResource.	Java	
Reliability	Avoid calling next() in hasNext().	Java	

Reliability	Avoid calling the setSize() from within componentResized().	Java	
Reliability	Avoid empty switch statements.	Java	
Reliability	Avoid returning null in a method declaration.	Java	
Maintainability	Avoid duplicate literals.	Java	
Maintainability	Avoid switch statements with few branches, instead use if-else.	Java	
Maintainability	A package does not depend on less stable packages.	Java	
Maintainability	Avoid cyclic dependencies between packages	Java	
Maintainability	Avoid using components calling too many other components.	Java	
Maintainability	Parent class does not reference any of its child classes.	Java	
Maintainability	Include a proper response code along with an appropriate return type.	Java	
Maintainability	Avoid using too long resource names.	Java	
Reliability	Use GenericEntity when returning a list of instances.	Java	
Maintainability	Use Java based configuration instead of XML based configuration.	Java	
Reliability	Use thread safe multi threading steps.	Java	
Maintainability	Avoid too deep steps hierarchy.	Java	
Maintainability	Avoid too deep jobs hierarchy.	Java	

Maintainability	Avoid hardcoding properties into XML descriptors, externalize them instead.	Java	
Maintainability	Avoid loading multiple XML configuration files.	Java	
Maintainability	Duplicated code: medium block	PHP	
Reliability	Else if statements should finish with an else clause.	PHP	
Reliability	Properly custom classes that extend from Exception class.	PHP	
Maintainability	Avoid too many choices in switch structures.	PHP	
Maintainability	Do not update control vars in 'for' loop body.	PHP	
Reliability	Avoid throwing exception base classes	PHP	
Maintainability	Avoid using too many statements in each case of a switch statement.	PHP	
Maintainability	Avoid using if-elseif-else chains.	PHP	

Anexos 4. Reglas de código de la herramienta PMD para el análisis de deuda técnica en lenguaje JAVA

Característica	Subcaracterística	Regla	Tipo de DT
Mantenibilidad	Comentarios	Comentario Obligatorio Tamaño del Comentario	Código
Mantenibilidad	Código innecesario	Variables locales no utilizadas Métodos privados no utilizados Parámetros sin usar	Código
Mantenibilidad	Nomenclatura estilo JavaBeans	Los miembros deben ser serailizados	Código
Confiabilidad	Básica	Bloque catch vacío Sentencia If vacía Sentencia while vacía Sentencia for vacía	Código
Confiabilidad	J2ee	MDBAndSessionBeanNamingConvention RemoteSessionInterfaceNamingConvention LocalInterfaceSessionNamingConvention LocalHomeNamingConvention RemoteInterfaceNamingConvention	Diseño
Confiabilidad	Diseño	PositionLiteralsFirstInComparisons ExcessiveImports UnnecessaryLocalBeforeReturn	Diseño
Confiabilidad	Tamaño del Código	Excesiva longitud Método Complejidad ciclomática Clases que tienen demasiados campos Excesiva longitud de clase	Código
Capacida de prueba	Prueba Unitaria por cada	JUnitStaticSuite JUnitSpelling JUnitTestsShouldIncludeAssert	Diseño
Capacidad de cambio	Acoplamiento	CouplingBetweenObjects ExcessiveImports LooseCoupling	Diseño

Anexos 5. Resultados de análisis por característica con la herramienta SonarQube

App	Caracteristica	Subcaracteristica	Evidencia	Num Evidencias	Tiempo(minutos)	Valor	Prioridad
App1	Confiabilidad	Arquitectura	Remove usage of generic wildcard type.	1	20	20	Mayor
					Total	20	
					Horas	0,33	
					DT	0,0	
App2	Changeability	Architecture	Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"	13	10	130	Mayor
	Changeability	Logic	Duplicated blocks	2	60	120	Mayor
	Changeability	Architecture	Class variable fields should not have public accessibility	1	10	10	Menor
	Changeability	Data	Magic numbers should not be used	236	5	1180	Menor
	Changeability	Logic	Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply	3	20	60	Menor
	Maintainability	Understandability	Unused method parameters should be removed	25	5	125	Mayor
	Maintainability	Readability	Package names should comply with a naming convention	12	20	240	Mayor
	Maintainability	Understandability	Unused local variables should be removed	8	5	40	Mayor
	Maintainability	Readability	Right curly brace and next "else", "catch" and "finally" keywords should be located on the same line	8	1	8	Mayor
	Maintainability	Readability	if/else/for/while/do statements should always use curly braces	1	10	10	Mayor

	Maintainability	Readability	The members of an interface declaration or class should appear in a pre-defined order	89	1	89	Menor
	Maintainability	Readability	Array designators "[]" should be on the type, not the variable	13	5	65	Menor
	Maintainability	Readability	Useless imports should be removed	1	10	10	Menor
	Maintainability	Understandability	TODO tags should be handled	2	20	40	Info
	Reliability	Data	public static fields should always be constant	1	20	20	Mayor
	Reliability	Data	String literals should not be duplicated	16	10	160	Menor
	Reliability	Exception handling	Exception handlers should preserve the original exception	8	10	80	Mayor
	Reliability	Logic	Nested blocks of code should not be left empty	3	20	60	Mayor
	Reliability	Instruction	Strings literals should be placed on the left side when checking for equality	3	10	30	Mayor
	Reliability	Logic	Methods should not be too complex	1	10	10	Mayor
					Total	2487	
					Horas	41,45	
					DT	**5,2**	
App3	Maintainability	Understandability	Unused local variables should be removed	64	5	320	Mayor
	Maintainability	Readability	Control structures should always use curly braces	17	2	34	Mayor
	Changeability	Architecture	Classes should not have too many methods	6	60	360	Mayor
	Reliability	Logic	&& and "||" should be used	5	5	25	Mayor
	Maintainability	Understandability	Sections of code should not be "commented out"	5	5	25	Mayor
	Maintainability	Understandability	Functions should not contain too	3	20	60	Mayor

			many return statements				
	Maintainability	Readability	Parentheses should not be used for calls to "echo"	2	2	4	Mayor
	Testability	Unit level	Functions should not have too many parameters	2	20	40	Mayor
	Reliability	Logic	Nested blocks of code should not be left empty	2	5	10	Mayor
	Reliability	Logic	Two branches in the same conditional structure should not have exactly the same implementation	1	10	10	Mayor
	Testability	Unit level	Functions should not be too complex	1	10	10	Mayor
	Maintainability	Understandability	Functions should not have too many lines	1	20	20	Mayor
	Reliability	Data	String literals should not be duplicated	117	10	1170	Menor
	Maintainability	Readability	Statements should be on separate lines	78	1	78	Menor
	Maintainability	Readability	Tabulation characters should not be used	70	2	140	Menor
	Maintainability	Readability	A close curly brace should be located at the beginning of a line	31	1	31	Menor
	Maintainability	Readability	Class names should comply with a naming convention	6	20	120	Menor
	Maintainability	Understandability	Literal boolean values should not be used in condition expressions	4	10	40	Menor
	Reliability	Instruction	Empty statements should be removed	3	2	6	Menor
					Total	2503	
					Horas	41,72	
					DT	**5,2**	
App4	Maintainability	Understandability	Unused method parameters should be removed	1	5	5	Mayor

	Reliability	Architecture	Generic wildcard types should not be used in return parameters	1	20	20	Mayor
	Maintainability	Understandability	TODO tags should be handled	1	20	20	Info
					Total	45	
					Horas	0,75	
					DT	**0,1**	
App5	Maintainability	Understandability	Sections of code should not be "commented out"	17	5	85	Mayor
	Reliability	Logic	&& and "||" should be used	10	5	50	Mayor
	Maintainability	Readability	Control structures should always use curly braces	9	2	18	Mayor
	Maintainability	Understandability	Unused local variables should be removed	3	5	15	Mayor
	Reliability	Logic	Nested blocks of code should not be left empty	2	5	10	Mayor
	Maintainability	Understandability	Functions should not have too many lines	1	20	20	Mayor
	Reliability	Data	String literals should not be duplicated	50	10	500	Menor
	Maintainability	Readability	Tabulation characters should not be used	22	2	44	Menor
	Maintainability	Readability	Statements should be on separate lines	20	1	20	Menor
	Maintainability	Understandability	Overriding methods should do more than simply call the same method in the super class	5	10	50	Menor
	Maintainability	Readability	Class names should comply with a naming convention	2	20	40	Menor
	Maintainability	Readability	Comments should not be located at the end of lines of code	5	1	5	Info
					Total	95	
					Horas	1,58	
					DT	**0,2**	

App6	Maintainability	Readability	Function names should comply with a naming convention	10	10	100	Mayor
	Maintainability	Readability	Useless parentheses around expressions should be removed to prevent any misunderstanding	2	1	2	Mayor
	Testability	Unit level	Functions should not be too complex	1	10	10	Mayor
	Changeability	Architecture	IP addresses should not be hardcoded	1	30	30	Mayor
	Maintainability	Readability	Local variable and function parameter names should comply with a naming convention	7	2	14	Menor
					Total	54	
					Horas	0,90	
					DT	**0,1**	
App7	Maintainability	Readability	Package names should comply with a naming convention	17	20	340	Mayor
	Maintainability	Understandability	Unused method parameters should be removed	11	5	55	Mayor
	Maintainability	Understandability	Avoid commented-out lines of code	7	60	420	Mayor
	Changeability	Logic	Duplicated blocks	5	60	300	Mayor
	Reliability	Architecture	Method parameters, caught exceptions and foreach variables should not be reassigned	5	5	25	Mayor
	Reliability	Exception handling	Generic exceptions Error, RuntimeException, Throwable and Exception should never be thrown	3	20	60	Mayor
	Maintainability	Readability	Right curly brace and next "else", "catch" and "finally" keywords should be located on the same line	3	1	3	Mayor

	Reliability	Instruction	Empty statements should be removed	3	2	6	Mayor
	Maintainability	Readability	Collection.isEmpty() should be used to test for emptiness	2	2	4	Mayor
	Reliability	Exception handling	Exception classes should be immutable	2	30	60	Mayor
	Maintainability	Readability	Class names should comply with a naming convention	2	20	40	Mayor
	Maintainability	Understandability	Unused local variables should be removed	2	5	10	Mayor
	Maintainability	Readability	Useless parentheses around expressions should be removed to prevent any misunderstanding	2	1	2	Mayor
	Maintainability	Understandability	Utility classes should not have a public constructor	1	30	30	Mayor
	Changeability	Data	Magic numbers should not be used	149	5	745	Menor
	Maintainability	Readability	The members of an interface declaration or class should appear in a pre-defined order	38	1	38	Menor
	Maintainability	Readability	Useless imports should be removed	7	10	70	Menor
	Reliability	Data	String literals should not be duplicated	3	10	30	Menor
	Maintainability	Readability	Comments should not be located at the end of lines of code	2	1	2	Menor
	Maintainability	Readability	Array designators "[]" should be on the type, not the variable	1	5	5	Menor
	Maintainability	Understandability	TODO tags should be handled	11	20	220	Info
					Total	2465	
					Horas	41,08	
					DT	**5,1**	
App8	Reliability	Instruction	Strings literals should be placed on	20	10	200	Mayor

			the left side when checking for equality				
	Maintainability	Readability	Switch cases should not have too many lines	19	10	190	Mayor
	Maintainability	Understandability	Avoid commented-out lines of code	10	60	600	Mayor
	Maintainability	Readability	Local variable and method parameter names should comply with a naming convention	10	20	200	Mayor
	Reliability	Exception handling	Exception handlers should preserve the original exception	5	10	50	Mayor
	Maintainability	Understandability	Unused method parameters should be removed	4	5	20	Mayor
	Reliability	Logic	Nested blocks of code should not be left empty	3	5	15	Mayor
	Testability	Unit level	Methods should not be too complex	3	10	30	Mayor
	Changeability	Architecture	Class variable fields should not have public accessibility	2	10	20	Mayor
	Reliability	Data	public static fields should always be constant	2	20	40	Mayor
	Maintainability	Readability	Method names should comply with a naming convention	2	5	10	Mayor
	Maintainability	Readability	Left curly braces should be located at the end of lines of code	2	1	2	Mayor
	Reliability	Data	Local variables should not shadow class fields	2	30	60	Mayor
	Changeability	Logic	Duplicated blocks	213	60	12780	Mayor
	Reliability	Logic	Switch statements should end with a default case	1	5	5	Mayor
	Maintainability	Understandability	Unused local variables should be removed	1	5	5	Mayor
	Maintainability	Understandability	Loops should not contain more than a single "break" or	1	30	30	Mayor

			"continue" statement				
	Changeability	Architecture	Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"	1	10	10	Mayor
	Reliability	Architecture	Method parameters, caught exceptions and foreach variables should not be reassigned	1	5	5	Mayor
	Reliability	Architecture	Methods should not be empty	1	5	5	Mayor
	Maintainability	Understandability	Utility classes should not have a public constructor	1	30	30	Mayor
	Maintainability	Readability	Field names should comply with a naming convention	1	2	2	Mayor
	Maintainability	Readability	Class names should comply with a naming convention	1	20	20	Mayor
	Reliability	Logic	Loop counters should not be assigned to from within the loop body	1	30	30	Mayor
	Changeability	Data	Magic numbers should not be used	118	5	590	Menor
	Maintainability	Readability	The members of an interface declaration or class should appear in a pre-defined order	11	1	11	Menor
	Maintainability	Readability	Array designators "[]" should be on the type, not the variable	8	5	40	Menor
	Changeability	Logic	Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply	6	10	60	Menor
	Maintainability	Readability	Useless imports should be removed	6	10	60	Menor

	Maintainability	Readability	Comments should not be located at the end of lines of code	2	1	2	Menor
	Maintainability	Readability	Tabulation characters should not be used	1	2	2	Menor
	Maintainability	Understandability	Literal boolean values should not be used in condition expressions	1	10	10	Menor
	Maintainability	Understandability	TODO tags should be handled	3	20	60	Info
					Total	15194	
					Horas	253,23	
					DT	**31,7**	

Anexos 6. Resultados de análisis por característica con la herramienta Kiuwan

App	Archivos	Violaciones	Reglas	Prioridad	Caracteristica	DT
App 1	11	11	Never ever use the database identifier in equals() and hashCode()		Reliability	5h 30
	1	1	Maximum allowed number of methods.		Maintainability	8h 00
	1	1	Avoid catch blocks with empty bodies.		Reliability	30m
	14	14	Follow the limit for number of statements in a method.		Maintainability	56h 00
	12	12	Always check object type before cast.		Reliability	6h 00
	1	11	Do not define 'Object' variables		Maintainability	1h 06
	11	11	Avoid unused fields.		Reliability	33m
	3	9	Do not make useless method overwriten.		Maintainability	27m
	4	4	Do not declare private fields that are used in only one method.		Maintainability	16h 00
	3	3	Avoid using components calling too many other components.		Maintainability	1h 30
	1	1	Avoid unused private fields.		Reliability	06m
	1	1	A package does not depend on less stable packages.		Maintainability	30m
	1	1	Avoid an excessive number of classes per package/namespace		Maintainability	4h 00
	14	25	Use constructors than initialize all the fields in a class.		Reliability	12h 30
	18	18	Provide Javadoc comments for public methods.		Maintainability	1h 48
	11	11	Follow the limit for number of return statements.		Maintainability	44h 00
	1	1	Use a branches block for 'else' statements.		Reliability	06m
	1	1	Avoid Exception, RuntimeException o Throwable in catch statements.		Reliability	30m

	1	1	Provide Javadoc comments for protected fields.		Maintainability	03m
	1	1	Avoid cyclic dependencies between packages		Maintainability	30m
	1	1	Provide a branch block for 'if' statements.		Maintainability	30m
	1	1	Only declare 'protected' constructors for abstract classes.		Maintainability	30m
	1	1	Do not use instanceof to distinguish between exceptions.		Reliability	30m
	12	12	Avoid incorrect name format in the final fields.		Maintainability	36m
	2	2	Avoid class names that are less than 5 characters.		Maintainability	06m
	1	1	Always declare the EntityBeans static fields as final.		Reliability	06m
	1	1	Provide Javadoc comments for private methods.		Maintainability	03m
	1	1	Declare fields with names in capital letters as final.		Maintainability	03m
	1	1	Avoid incorrect name format in non-final static fields.		Maintainability	03m
	1	1	Leave a white space between arguments		Maintainability	03m
		Total DT	162h			
App 2	1	1	Cyclomatic complexity.		Maintainability	4h 00
	1	3	Close input and output resources in finally blocks.		Reliability	1h 30
	1	3	Avoid catch blocks with empty bodies.		Reliability	1h 30
	1	1	Avoid parameter method names that provoke conflicts with class members names.		Maintainability	03m
	1	1	Maximum allowed number of methods.		Maintainability	8h 00
	1	1	Provide Javadoc comments for public fields.		Maintainability	06m
	1	1	Avoid unused imports		Maintainability	03m
	5	155	Avoid using numeric literals.		Maintainability	15h 30

	4	48	Do not declare private fields that are used in only one method.		Maintainability	192 h
	6	37	Follow the limit for number of statements in a method.		Maintainability	148 h
	4	15	Avoid calling this.getClass.getResource.		Reliability	7h 30
	2	12	Declare arrays with [] braces after the array type and before the variable name(s).		Maintainability	1h 12
	4	11	Avoid duplicate literals.		Maintainability	5h 30
	4	9	Always check object type before cast.		Reliability	4h 30
	3	8	Declare the List and Set variables with their interface type.		Maintainability	48m
	2	6	Initialize all local variables at the declaration statement.		Reliability	36m
	5	5	Avoid using components calling too many other components.		Maintainability	2h 30
	5	5	Classes internally strongly coupled must be avoided.		Maintainability	20h 00
	4	4	Define class attributes at the beginning of the class.		Maintainability	12m
	4	4	Follow the limit for number of lines in a method.		Maintainability	16h 00
	2	3	Avoid calling varString.equals('literal') or varString.equalsIgnoreCase('literal').		Reliability	09m
	1	2	Duplicated code: small block		Maintainability	12m
	1	1	Avoid non-final public static fields.		Reliability	4h 00
	1	1	A package does not depend on less stable packages.		Maintainability	30m
	1	1	Avoid fields and methods with the same name.		Maintainability	03m
	1	1	Too many uses of the negation operator (!) in a method.		Maintainability	30m
	10	12	Use constructors than initialize all the fields in a class.		Reliability	6h 00
	11	11	Provide Javadoc comments for public methods.		Maintainability	1h 06

	1	8	Avoid using variables with the same name.		Maintainability	24m
	1	4	Avoid assignments in while / do-while loop condition.		Reliability	2h 00
	1	1	Avoid Exception, RuntimeException o Throwable in catch statements.		Reliability	30m
	1	1	Avoid capturing java.lang.Exception exceptions.		Reliability	30m
	1	1	Do not use return statements inside catch blocks.		Reliability	30m
	1	1	Provide Javadoc comments for protected fields.		Maintainability	03m
	1	1	Follow the limit for number of return statements.		Maintainability	4h 00
	1	1	Avoid cyclic dependencies between packages		Maintainability	30m
	1	1	Define every field private or protected		Maintainability	03m
	1	1	Provide a branch block for 'if' statements.		Maintainability	30m
	4	4	Provide Javadoc comments for private methods.		Maintainability	12m
	1	3	Leave a white space between arguments		Maintainability	09m
	1	1	Always declare the EntityBeans static fields as final.		Reliability	06m
	1	1	Justify the opening and closing braces in blocks.		Maintainability	03m
		Total DT	451h			
App 3	6	36	Avoid using echo or print to construct HTML.		Maintainability	3h 36
	12	13	Do not use exit() or die() for error processing		Maintainability	52h 00
	1	1	Avoid calling a function or method with more parameters than declared.		Reliability	06m
	7	81	Avoid unused local variables.		Reliability	4h 03
	4	4	Maximum allowed number of methods.		Maintainability	32h 00
	2	2	Too many parameters in function.		Maintainability	8h 00

	1	1	Do not use include and its variants with parentheses		Reliability	06m
	1	1	Avoid unused function parameters.		Reliability	03m
	44	257	Use simple quotes to demarcate string literals, except when containing apostrophes, escaped chars or variable substitutions		Maintainability	25h 42
	12	83	Avoid literals in method calls. Named constants make source code easier to understand and maintain.		Maintainability	332 h
	24	60	Do not use if ($var) to check if a variable is initialized.		Reliability	6h 00
	3	6	Duplicated code: small block		Maintainability	36m
	3	3	Variable initialization.		Reliability	18m
	2	2	Avoid functions and methods with too many lines of code		Maintainability	8h 00
	2	2	Avoid classes with too many methods		Maintainability	8h 00
	1	1	Avoid classes with too many lines of code		Maintainability	4h 00
	43	142	Use strict comparisons.		Reliability	14h 12
	12	12	PHP files declaring global symbols (functions, classes) should not have side effects		Maintainability	96h 00
	1	2	Use of a string created just from a variable.		Maintainability	12m
	2	2	Else if statements should finish with an else clause.		Reliability	1h 00
		Total DT	595h			
App 4	1	1	Never ever use the database identifier in equals() and hashCode()		Reliability	30m
	2	3	Follow the limit for number of statements in a method.		Maintainability	12h 00
	1	3	Do not make useless method overwriten.		Maintainability	09m
	2	2	Always check object type before cast.		Reliability	1h 00
	1	1	Do not define 'Object' variables		Maintainability	06m
	1	1	Avoid unused fields.		Reliability	03m

	1	1	Do not declare private fields that are used in only one method.		Maintainability	4h 00
	1	1	Avoid using components calling too many other components.		Maintainability	30m
	1	1	Avoid multiple comparisons within if.		Maintainability	30m
	1	1	Avoid TODO comments in production code.		Reliability	03m
	4	4	Provide Javadoc comments for public methods.		Maintainability	24m
	2	3	Use constructors than initialize all the fields in a class.		Reliability	1h 30
	1	1	Follow the limit for number of return statements.		Maintainability	4h 00
	1	1	Only declare 'protected' constructors for abstract classes.		Maintainability	30m
	1	2	With multiple (overloaded) constructors, use 'this' to call more generic constructors inside constructors.		Maintainability	12m
	1	1	Avoid incorrect name format in the final fields.		Maintainability	03m
		Total DT	25h 30m			
App 5	5	101	Avoid using echo or print to construct HTML.		Maintainability	10h 06
	17	17	Do not use exit() or die() for error processing		Maintainability	68h 00
	2	3	Avoid unused local variables.		Reliability	09m
	2	2	Avoid unused function parameters.		Reliability	06m
	16	188	Use simple quotes to demarcate string literals, except when containing apostrophes, escaped chars or variable substitutions		Maintainability	18h 48
	17	61	Avoid literals in method calls. Named constants make source code easier to understand and maintain.		Maintainability	244 h
	5	6	Variable initialization.		Reliability	36m

	1	1	Avoid functions and methods with too many lines of code		Maintainability	4h 00
	1	1	Avoid using if-elseif-else chains.		Maintainability	30m
	14	61	Use strict comparisons.		Reliability	6h 06
	17	17	PHP files declaring global symbols (functions, classes) should not have side effects		Maintainability	136 h
	2	8	Else if statements should finish with an else clause.		Reliability	4h 00
	3	3	Omit closing tag in files with only php code		Reliability	18m
	2	2	Use of a string created just from a variable.		Maintainability	12m
		Total DT	492h			
App 7	3	3	Never ever use the database identifier in equals() and hashCode()		Reliability	1h 30
	6	6	Provide Javadoc comments for public classes and interfaces.		Maintainability	3h 00
	4	4	Avoid unused imports		Maintainability	12m
	2	75	Avoid using numeric literals.		Maintainability	7h 30
	9	29	Follow the limit for number of statements in a method.		Maintainability	116 h
	1	24	Do not declare private fields that are used in only one method.		Maintainability	96h 00
	6	6	Always check object type before cast.		Reliability	3h 00
	6	6	Avoid using components calling too many other components.		Maintainability	3h 00
	4	5	Avoid modifications on method or constructor parameters.		Reliability	20h 00
	3	3	Avoid unused fields.		Reliability	09m
	3	3	Initialize all local variables at the declaration statement.		Reliability	18m
	3	3	Avoid multiple comparisons within if.		Maintainability	1h 30
	3	3	Avoid TODO comments in production code.		Reliability	09m

	2	2	Avoid unused local variables.		Reliability	12m
	2	2	Naming convention for class names.		Maintainability	06m
	1	1	Avoid duplicate literals.		Maintainability	30m
	1	1	Classes internally strongly coupled must be avoided.		Maintainability	4h 00
	1	1	Define class attributes at the beginning of the class.		Maintainability	03m
	1	1	Follow the limit for number of lines in a method.		Maintainability	4h 00
	1	1	Getter methods conventions.		Maintainability	03m
	17	17	Provide Javadoc comments for public methods.		Maintainability	1h 42
	5	11	Use constructors than initialize all the fields in a class.		Reliability	5h 30
	9	9	Avoid class or interface names too long.		Maintainability	27m
	3	6	Avoid Exception, RuntimeException o Throwable in catch statements.		Reliability	3h 00
	4	4	Provide Javadoc comments for protected fields.		Maintainability	12m
	3	3	Avoid capturing java.lang.Exception exceptions.		Reliability	1h 30
	3	3	Follow the limit for number of return statements.		Maintainability	12h 00
	3	3	Write one statement per line.		Maintainability	09m
	3	3	Serialization of fields of a 'Bean' class.		Reliability	12h 00
	1	1	Provide a by default private constructor in utility classes.		Maintainability	4h 00
	7	17	With multiple (overloaded) constructors, use 'this' to call more generic constructors inside constructors.		Maintainability	1h 42
	3	3	Avoid incorrect name format in the final fields.		Maintainability	09m
	3	3	Provide Javadoc comments for private methods.		Maintainability	09m
	3	3	Justify the opening and closing braces in blocks.		Maintainability	09m

	3	3	Serializable classes should always have a final static serialVersionUID field.		Reliability	09m
	2	2	Always start a class name with a capital letter.		Maintainability	06m
		Total DT	304h			
App 8	2	5	Cyclomatic complexity.		Maintainability	20h 00
	1	4	Avoid duplicated imports		Maintainability	12m
	2	3	Avoid catch blocks with empty bodies.		Reliability	1h 30
	1	1	Provide Javadoc comments for public fields.		Maintainability	06m
	1	1	Avoid unused private methods and constructors.		Maintainability	03m
	1	1	Provide Javadoc comments for public classes and interfaces.		Maintainability	30m
	1	1	Avoid nested IF sentences with too many levels.		Maintainability	30m
	1	1	Avoid unused imports		Maintainability	03m
	1	1	Avoid calling non-final, non-static and non-private methods from constructors.		Reliability	30m
	1	1	Do not assign loop control variables in the body of a for loop.		Reliability	30m
	1	1	Provide 'default' label for each switch statement.		Reliability	30m
	1	1	Provide a break or return statement for the default label in a switch.		Reliability	06m
	2	30	Avoid using numeric literals.		Maintainability	3h 00
	1	19	Avoid calling varString.equals('literal') or varString.equalsIgnoreCase('literal').		Reliability	57m
	2	13	Follow the limit for number of statements in a method.		Maintainability	52h 00
	1	10	Avoid duplicate literals.		Maintainability	5h 00
	1	7	Do not declare private fields that are used in only one method.		Maintainability	28h 00

	2	7	Declare arrays with [] braces after the array type and before the variable name(s).		Maintainability	42m
	2	3	Initialize all local variables at the declaration statement.		Reliability	18m
	2	3	Follow the limit for number of lines in a method.		Maintainability	12h 00
	1	2	Avoid non-final public static fields.		Reliability	8h 00
	1	2	Avoid giving method local variables and parameters the same name as class fields.		Maintainability	06m
	2	2	Avoid using components calling too many other components.		Maintainability	1h 00
	1	2	Capture OutOfMemoryError for large arrays initializations.		Reliability	1h 00
	1	1	Avoid modifications on method or constructor parameters.		Reliability	4h 00
	1	1	Avoid unused local variables.		Reliability	06m
	1	1	Classes internally strongly coupled must be avoided.		Maintainability	4h 00
	1	1	Avoid usage of * in import statements.		Maintainability	03m
	1	1	Declare the List and Set variables with their interface type.		Maintainability	06m
	1	1	Naming convention for class names.		Maintainability	03m
	1	1	Define class attributes at the beginning of the class.		Maintainability	03m
	1	1	Declare classes where all constructors are private and 'final'.		Reliability	8h 00
	2	3	Avoid Exception, RuntimeException o Throwable in catch statements.		Reliability	1h 30
	2	3	Avoid capturing java.lang.Exception exceptions.		Reliability	1h 30
	2	2	Provide Javadoc comments for protected fields.		Maintainability	06m
	1	2	Define every field private or protected		Maintainability	06m

	1	2	Avoid the incorrect naming of non-static methods.		Maintainability	06m
	1	1	Use constructors than initialize all the fields in a class.		Reliability	30m
	1	1	Follow the limit for number of return statements.		Maintainability	4h 00
	1	1	Provide Javadoc comments for public methods.		Maintainability	06m
	1	1	Provide a by default private constructor in utility classes.		Maintainability	4h 00
	1	1	Avoid assigning a variable to itself		Maintainability	03m
	1	1	Avoid Empty Classes		Reliability	03m
	2	10	Avoid incorrect name format in local variables.		Maintainability	30m
	1	2	Always declare the EntityBeans static fields as final.		Reliability	12m
	2	2	Avoid using do-while statements.		Maintainability	1h 00
	1	2	Always follow the java method naming conventions.		Maintainability	06m
	1	1	Avoid incorrect name format in the non-static fields.		Maintainability	03m
	1	1	Avoid class names that are less than 5 characters.		Maintainability	03m
	1	**1**	**Provide Javadoc comments for private methods.**		**Maintainability**	**03m**
	1	1	Declare fields with names in capital letters as final.		Maintainability	03m
	1	1	Justify the opening and closing braces in blocks.		Maintainability	03m
	1	1	Avoid classes and fields with the same name.		Maintainability	03m
	1	1	Always start a class name with a capital letter.		Maintainability	03m
		Total DT	167h			

Anexos 7. Resultados de análisis por característica con la herramienta PMD

					App1			App2			App4			App7			App8		
Caracteristica	**Subcaracteristica**	**Rule**	**Remediacion**	**No-Remediacion**	**count**	**remediacion**	**no-remediacion**	**count**	**remediacion**	**no-remediacion**	**count**	**remediacion**	**no-remediacion**	**count**	**remediacion**	**no-remediacion**	**count**	**remediacion**	**no-remediacion**
Confiabilidad	Basic	EmptyCatchBlock	5	1	0	0	0	3	15	3	0	0	0	0	0	0	3	15	3
Confiabilidad	Basic	EmptyIfStmt	10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptyWhileStmt	10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptyTryBlock	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptyFinallyBlock	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptySwitchStatements	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	JumbledIncrementer	10	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	ForLoopShouldBeWhileLoop	20	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UnnecessaryConversionTemporary	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	OverrideBothEqualsAndHashcode	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	DoubleCheckedLocking	10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	ReturnFromFinallyBlock	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptySynchronizedBlock	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UnnecessaryReturn	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptyStaticInitializer	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UnconditionalIfStatement	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptyStatementNotInLoop	5	10	0	0	0	0	0	0	0	0	0	3	15	30	0	0	0
Confiabilidad	Basic	BooleanInstantiation	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UnnecessaryFinalModifier	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	CollapsibleIfStatements	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UselessOverridingMethod	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	ClassCastExceptionWithToArray	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	AvoidDecimalLiteralsInBigDecimalConstructor	5	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UselessOperationOnImmutable	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	MisplacedNullCheck	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	UnusedNullCheckInEquals	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	AvoidThreadGroup	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	BrokenNullCheck	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	BigIntegerInstantiation	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	AvoidUsingOctalValues	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	AvoidUsingHardCodedIP	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	CheckResultSet	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	AvoidMultipleUnaryOperators	10	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	EmptyInitializer	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Basic	DontCallThreadRun	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	UseProperClassLoader	10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	MDBAndSessionBeanNamingConvention	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	RemoteSessionInterfaceNamingConvention	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	LocalInterfaceSessionNamingConvention	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	LocalHomeNamingConvention	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	RemoteInterfaceNamingConvention	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	DoNotCallSystemExit	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	StaticEJBFieldShouldBeFinal	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	J2EE	DoNotUseThreads	5	10	0	0	0	1	5	10	0	0	0	1	5	10	1	5	10
Confiabilidad	Tipo de datos	LooseCoupling	10	1	0	0	0	8	80	8	0	0	0	0	0	0	1	10	1
Confiabilidad	Tipo de datos	CloneMethodMustImplementCloneable	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tipo de datos	UnusedImports	10	1	0	0	0	1	10	1	0	0	0	12	120	12	10	100	10
Confiabilidad	Tipo de datos	SignatureDeclareThrowsException	5	1	0	0	0	0	0	0	0	0	0	3	15	3	0	0	0
Confiabilidad	Tamaño de código	NPathComplexity	20	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	ExcessiveMethodLength	20	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	ExcessiveParameterList	20	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	ExcessiveClassLength	20	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	CyclomaticComplexity	10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	ExcessivePublicCount	20	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	TooManyFields	20	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	NcssMethodCount	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	NcssTypeCount	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	NcssConstructorCount	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Confiabilidad	Tamaño de código	TooManyMethods	20	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mantenibilidad	Comentarios	Comentario Obligatorio	5	1	332	1660	332	297	1485	297	33	165	33	193	965	193	39	195	39
Mantenibilidad	Comentarios	Tamaño del Comentario	5	1	0	0	0	6	30	6	1	5	1	6	30	6	4	20	4
Mantenibilidad	Comentarios	Contenido del Comentario	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mantenibilidad	Código innesesario	UnusedPrivateField	1	5	1	1	5	0	0	0	0	0	0	0	0	0	0	0	0
Mantenibilidad	Código innesesario	UnusedLocalVariable	1	5	0	0	0	0	0	0	0	0	0	2	2	10	1	1	5
Mantenibilidad	Código innesesario	UnusedPrivateMethod	5	5	0	0	0	0	0	0	0	0	0	0	0	0	1	5	5
Mantenibilidad	Código innesesario	UnusedFormalParameter	5	5	0	0	0	20	100	100	0	0	0	9	45	45	3	15	15
Mantenibilidad	Nomenclatura JavaBear	BeanMembersShouldSerialize	5	1	5	25	5	76	380	76	2	10	2	62	310	62	21	105	21
Mantenibilidad	Nomenclatura JavaBear	MissingSerialVersionUID	1	5	0	0	0	0	0	0	0	0	0	3	3	15	0	0	0
Mantenibilidad	Duplicacion	DuplicatedCode	10	10	10	100	100	3	30	30	0	0	0	2	20	20	18	180	180
Capacidad de prueba	Pruebas unitarias	JUnitStaticSuite	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de prueba	Pruebas unitarias	JUnitSpelling	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de prueba	Pruebas unitarias	JUnitAssertionsShouldIncludeMessage	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de prueba	Pruebas unitarias	JUnitTestsShouldIncludeAssert	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de prueba	Pruebas unitarias	TestClassWithoutTestCases	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de prueba	Pruebas unitarias	UnnecessaryBooleanAssertion	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	UseSingleton	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	SimplifyBooleanReturns	10	5	0	0	0	0	0	0	1	10	5	3	30	15	0	0	0
Capacidad de cambio	Diseño	SimplifyBooleanExpressions	5	10	0	0	0	0	0	0	0	0	0	0	0	0	1	5	10
Capacidad de cambio	Diseño	SwitchStmtsShouldHaveDefault	5	1	0	0	0	0	0	0	0	0	0	0	0	0	1	5	1
Capacidad de cambio	Diseño	AvoidDeeplyNestedIfStmts	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AvoidReassigningParameters	10	10	0	0	0	0	0	0	0	0	0	5	50	50	1	10	10
Capacidad de cambio	Diseño	SwitchDensity	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	ConstructorCallsOverridableMethod	10	5	0	0	0	0	0	0	0	0	0	0	0	0	1	10	5
Capacidad de cambio	Diseño	AccessorClassGeneration	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	FinalFieldCouldBeStatic	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	CloseResource	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	NonStaticInitializer	1	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	DefaultLabelNotLastInSwitchStmt	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	NonCaseLabelInSwitchStatement	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	OptimizableToArrayCall	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	BadComparison	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	EqualsNull	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	ConfusingTernary	5	10	11	55	110	5	25	50	1	5	10	3	15	30	0	0	0
Capacidad de cambio	Diseño	InstantiationToGetClass	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	IdempotentOperations	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Capacidad de cambio	Diseño	SimpleDateFormatNeedsLocale	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	ImmutableField	5	5	0	0	0	6	30	30	1	5	5	2	10	10	0	0	0
Capacidad de cambio	Diseño	UseLocaleWithCaseConversions	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AvoidProtectedFieldInFinalClass	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AssignmentToNonFinalStatic	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	MissingStaticMethodInNonInstantiatableClass	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AvoidSynchronizedAtMethodLevel	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	MissingBreakInSwitch	5	5	0	0	0	0	0	0	0	0	0	0	0	0	1	5	5
Capacidad de cambio	Diseño	UseNotifyAllInsteadOfNotify	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AvoidInstanceofChecksInCatchClause	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AbstractClassWithoutAbstractMethod	5	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	SimplifyConditional	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	CompareObjectsWithEquals	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	PositionLiteralsFirstInComparisons	5	1	0	0	0	0	0	0	0	0	0	0	0	0	20	100	20
Capacidad de cambio	Diseño	UnnecessaryLocalBeforeReturn	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	NonThreadSafeSingleton	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	UncommentedEmptyMethod	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	UncommentedEmptyConstructor	5	10	11	55	110	5	25	50	1	5	10	4	20	40	0	0	0
Capacidad de cambio	Diseño	AvoidConstantsInterface	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	UnsynchronizedStaticDateFormatter	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	PreserveStackTrace	5	10	0	0	0	0	0	0	0	0	0	3	15	30	0	0	0
Capacidad de cambio	Diseño	UseCollectionIsEmpty	1	10	0	0	0	0	0	0	0	0	0	2	2	20	0	0	0
Capacidad de cambio	Diseño	ClassWithOnlyPrivateConstructorsShouldBeF	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	EmptyMethodInAbstractClassShouldBeAbstra	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	SingularField	1	5	1	1	5	48	48	240	0	0	0	24	24	120	7	7	35
Capacidad de cambio	Diseño	ReturnEmptyArrayRatherThanNull	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	AbstractClassWithoutAnyMethod	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Diseño	TooFewBranchesForASwitchStatement	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Acoplamiento	CouplingBetweenObjects	10	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Acoplamiento	ExcessiveImports	10	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Capacidad de cambio	Acoplamiento	LooseCoupling	5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
				Total	371	1897	667	479	2263	901	40	205	66	342	1696	721	135	794	380
				Confiabilidad	0	0	0	13	110	22	0	0	0	19	155	55	15	130	24
				Mantenibilidad	348	1786	442	402	2025	509	36	180	36	277	1375	351	87	521	269
				Capacidad de prue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
				Capacidad de cam	23	111	225	64	128	370	4	25	30	46	166	315	33	143	87
				Ratio	2,84407796			2,51165372			3,10606061			2,35228849			2,08947368		

Deuda Técnica en horas

Mantenibilidad		29,76666667			33,75			3,00			22,92			8,68	
Confiabilidad		0			1,83			0,00			2,58			2,17	
Capacidad de cambio		1,85			2,13			0,42			2,77			2,38	
Capacidad de prueba		0			0,00			0,00			0,00			0,00	
Total DT		31,61666667			37,72			3,42			28,27			13,23	
DT dias/hombre		3,952083333			4,71458333			0,42708333			3,533333333			1,654166667	

Anexos 8. : Violaciones de código a corregir

Herramienta	Característica	Subcaracteristica	Regla	CR	Severidad	Lenguaje	Repeticiones
SonarQube	Maintainability	Understandability	Sections of code should not be "commented out"	5	Mayor	PHP	1
	Reliability	Data	String literals should not be duplicated	10	Menor	PHP	1
	Maintainability	Readability	Function names should comply with a naming convention	10	Mayor	Python	1
	Maintainability	Readability	Package names should comply with a naming convention	20	Mayor	Java	1
	Maintainability	Understandability	Unused method parameters should be removed	5	Mayor	Java	1
	Maintainability	Understandability	Avoid commented-out lines of code	60	Mayor	Java	2
	Changeability	Logic	Duplicated blocks	60	Mayor	Java	2
	Reliability	Exception handling	Generic exceptions Error, RuntimeException, Throwable and Exception should never be thrown	20	Mayor	Java	1
	Reliability	Exception handling	Exception classes should be immutable	30	Mayor	Java	1
	Changeability	Data	Magic numbers should not be used	5	Menor	Java	2
	Maintainability	Readability	Useless imports should be removed	10	Menor	Java	2
	Maintainability	Understandability	TODO tags should be handled	20	Info	Java	2
	Reliability	Instruction	Strings literals should be placed on the left side when checking for equality	10	Mayor	Java	1
	Maintainability	Readability	Switch cases should not have too many lines	10	Mayor	Java	1

	Maintainability	Readability	Local variable and method parameter names should comply with a naming convention	20	Mayor	Java	1
	Reliability	Data	Local variables should not shadow class fields	30	Mayor	Java	1
	Changeability	Logic	Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply	10	Menor	Java	1
Kiuwan	Maintainability		Follow the limit for number of statements in a method.	240		Java	3
	Reliability		Always check object type before cast.	30		Java	2
	Maintainability		Do not declare private fields that are used in only one method.	240		Java	3
	Reliability		Use constructors than initialize all the fields in a class.	30		Java	2
	Maintainability		Follow the limit for number of return statements.	240		Java	3
	Maintainability		Avoid using echo or print to construct HTML.	6		PHP	1
	Maintainability		Do not use exit() or die() for error processing	240		PHP	1
	Maintainability		Use simple quotes to demarcate string literals, except when containing apostrophes, escaped chars or variable substitutions	6		PHP	1

	Maintainability		Avoid literals in method calls. Named constants make source code easier to understand and maintain.	240		PHP	1
	Maintainability		Avoid functions and methods with too many lines of code	240		PHP	1
	Reliability		Use strict comparisons.	6		PHP	1
	Maintainability		PHP files declaring global symbols (functions, classes) should not have side effects	480		PHP	1
	Reliability		Else if statements should finish with an else clause.	30		PHP	1
	Reliability		Never ever use the database identifier in equals() and hashCode()	30		Java	1
	Maintainability		Provide Javadoc comments for public classes and interfaces.	30		Java	1
	Maintainability		Avoid using numeric literals.	6		Java	2
	Maintainability		Avoid using components calling too many other components.	30		Java	2
	Reliability		Avoid modifications on method or constructor parameters.	240		Java	2
	Maintainability		Avoid multiple comparisons within if.	30		Java	1

	Maintainability		Classes internally strongly coupled must be avoided.	240		Java	2
	Maintainability		Follow the limit for number of lines in a method.	240		Java	2
	Maintainability		Provide Javadoc comments for public methods.	6		Java	1
	Reliability		Avoid Exception, RuntimeException o Throwable in catch statements.	30		Java	2
	Reliability		Avoid capturing java.lang.Exception exceptions.	30		Java	2
	Reliability		Serialization of fields of a 'Bean' class.	240		Java	1
	Maintainability		Provide a by default private constructor in utility classes.	240		Java	2
	Maintainability		With multiple (overloaded) constructors, use 'this' to call more generic constructors inside constructors.	6		Java	1
	Maintainability		Cyclomatic complexity.	240		Java	1
	Reliability		Avoid catch blocks with empty bodies.	30		Java	1
	Reliability		Avoid calling varString.equals('literal') or varString.equalsIgnoreCase('literal').	3		Java	1
	Maintainability		Avoid duplicate literals.	30		Java	1

	Reliability		Avoid non-final public static fields.	240		Java	1
	Reliability		Capture OutOfMemoryError for large arrays initializations.	30		Java	1
	Reliability		Declare classes where all constructors are private and 'final'.	480		Java	1
	Maintainability		Avoid using do-while statements.	30		Java	1
PMD	Confiabilidad	Tipo de datos	UnusedImports	10	1	Java	2
	Mantenibilidad	Comentarios	Comentario Obligatorio	5	1	Java	3
	Mantenibilidad	Nomenclatura JavaBeans	BeanMembersShouldSerialize	5	1	Java	2
	Mantenibilidad	Duplicacion	DuplicatedCode	10	10	Java	1
	Capacidad de cambio	Diseño	PositionLiteralsFirstInComparisons	5	1	Java	1

BIBLIOGRAFÍA

Allman, E. (2012). Managing Technical Debt. *Commun. ACM*, *55*(5), 50–55. http://doi.org/10.1145/2160718.2160733

Alzaghoul, E. (2013). Economics-driven Approach for Managing Technical Debt in Cloud-Based Architectures. http://doi.org/10.1109/UCC.2013.49

Alzaghoul, E. (2014). Evaluating Technical Debt in Cloud-based Architectures using Real Options. http://doi.org/10.1109/ASWEC.2014.27

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2015). The financial aspect of managing technical debt : A systematic literature review. *Information and Software Technology*, *64*, 52–73. http://doi.org/10.1016/j.infsof.2015.04.001

Bellomo, S., Nord, R. L., & Ozkaya, I. (n.d.). A Study of Enabling Factors for Rapid Fielding Combined Practices to Balance Speed and Stability.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., … Zazworka, N. (2010). Managing Technical Debt in Software-reliant Systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (pp. 47–52). New York, NY, USA: ACM. http://doi.org/10.1145/1882362.1882373

Cast Software. (n.d.). Retrieved from http://www.castsoftware.com/

Cunningham, W. (1992). The WyCash Portfolio Management System. *SIGPLAN OOPS Mess.*, *4*(2), 29–30. http://doi.org/10.1145/157710.157715

Curtis, B., Sappidi, J., & Szynkarski, A. (2012a). Estimating the Principal of an Application's Technical Debt. *IEEE Software*, *29*(6), 34–42. http://doi.org/10.1109/MS.2012.156

Curtis, B., Sappidi, J., & Szynkarski, A. (2012b). Estimating the size, cost, and types of Technical Debt. In *Managing Technical Debt (MTD), 2012 Third International Workshop on* (pp. 49–53). http://doi.org/10.1109/MTD.2012.6226000

Díaz, G., & Bermejo, J. R. (2013). Static analysis of source code security: Assessment of tools against {SAMATE} tests. *Information and Software Technology*, *55*(8), 1462–1476. http://doi.org/http://dx.doi.org/10.1016/j.infsof.2013.02.005

Eisenberg, R. J. (2012). A Threshold Based Approach to Technical Debt. *SIGSOFT Softw. Eng. Notes*, *37*(2), 1–6. http://doi.org/10.1145/2108144.2108151

Ernst, N. A. (2012). On the role of requirements in understanding and managing

technical debt. In *Managing Technical Debt (MTD), 2012 Third International Workshop on* (pp. 61–64). http://doi.org/10.1109/MTD.2012.6226002

Fern, C., & Garbajosa, J. (2015). An Analysis of Techniques and Methods for Technical Debt Management : a Reflection from the Architecture Perspective. http://doi.org/10.1109/SAM.2015.11

FindBugs. (n.d.). Retrieved from http://findbugs.sourceforge.net/

Fontana, F. A., Ferme, V., & Zanoni, M. (2015). Towards assessing software architecture quality by exploiting code smell relations. http://doi.org/10.1109/SAM.2015.8

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. *Addison-Wesley Professional* (1ra ed.).

Griffith, I., & Izurieta, C. (n.d.). Design Pattern Decay : The Case for Class Grime, 1–4.

Guo, Y., Spínola, R. O., & Seaman, C. (2016). Exploring the costs of technical debt management -- a case study. *Empirical Software Engineering*, *21*(1), 159–182. http://doi.org/10.1007/s10664-014-9351-7

Heidenberg, J., & Porres, I. (2010). Metrics Functions for Kanban Guards. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on* (pp. 306–310). http://doi.org/10.1109/ECBS.2010.43

Izurieta, C., & Bieman, J. M. (2008). Testing Consequences of Grime Buildup in Object Oriented Design Patterns.

Kazman, R., Cai, Y., Mo, R., Feng, Q., & Xiao, L. (2015). A Case Study in Locating the Architectural Roots of Technical Debt, 179–188. http://doi.org/10.1109/ICSE.2015.146

Kiuwan. (n.d.). Retrieved from https://www.kiuwan.com/es/

Kruchten, P. (2012). Strategic Management of Technical Debt Tutorial Synopsis, 0–2. http://doi.org/10.1109/QSIC.2012.17

Kuipers, T. (2011). An Empirical Model of Technical Debt and Interest Software Improvement Group.

Lattix. (n.d.). Retrieved from http://lattix.com/

Letouzey, J. L. (2012). The SQALE method for evaluating Technical Debt. In *Managing Technical Debt (MTD), 2012 Third International Workshop on* (pp. 31–

36). http://doi.org/10.1109/MTD.2012.6225997

Letouzey, J. L., & Coq, T. (2010). The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code. In *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on* (pp. 43–48). http://doi.org/10.1109/VALID.2010.31

Letouzey, J. L., & Ilkiewicz, M. (2012). Managing Technical Debt with the SQALE Method. *IEEE Software*, *29*(6), 44–51. http://doi.org/10.1109/MS.2012.129

Letouzey, J.-L. (2012). *The SQALE Method*.

Li, Z. (2015). Architectural Technical Debt Identification based on Architecture Decisions and Change Scenarios, (895528), 65–74. http://doi.org/10.1109/WICSA.2015.19

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193–220. http://doi.org/http://dx.doi.org/10.1016/j.jss.2014.12.027

Li, Z., Liang, P., & Avgeriou, P. (2014). Architectural Debt Management in Value-Oriented Architecting 1 9.

Mamun, A. Al, & Berger, C. (2014). Explicating , Understanding and Managing Technical Debt from Self-Driving Miniature Car Projects, *1*. http://doi.org/10.1109/MTD.2014.15

Martini, A., Bosch, J., & Chaudron, M. (2014). Architecture Technical Debt: Understanding Causes and a Qualitative Model. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications* (pp. 85–92). http://doi.org/10.1109/SEAA.2014.65

NDepend. (n.d.). Retrieved from http://www.ndepend.com/

Nord, R. L., Ozkaya, I., Kruchten, P., & Gonzalez-Rojas, M. (2012). In Search of a Metric for Managing Architectural Technical Debt. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on* (pp. 91–100). http://doi.org/10.1109/WICSA-ECSA.212.17

Seaman, C. (2013). Measuring and Monitoring Technical Debt, (March).

Seaman, C., & Guo, Y. (2011). Chapter 2 - Measuring and Monitoring Technical Debt. In M. V Zelkowitz (Ed.), (Vol. 82, pp. 25–46). Elsevier.

http://doi.org/http://dx.doi.org/10.1016/B978-0-12-385512-1.00002-5

Seaman, C., Guo, Y., Izurieta, C., Zazworka, N., Shull, F., & Vetrò, A. (2012). Using Technical Debt Data in Decision Making : Potential Decision Approaches, 45–48.

SonarGraph. (n.d.). Retrieved from https://www.hello2morrow.com/products/sonargraph

SonarQube. (n.d.). Retrieved from http://www.sonarqube.org/

SourceMeter. (n.d.). Retrieved from https://www.sourcemeter.com/

SourceMonitor. (n.d.). Retrieved from http://www.campwoodsw.com/sourcemonitor.html

Squoring. (n.d.). Retrieved from http://www.squoring.com/en/

Structure101. (n.d.). Retrieved from http://structure101.com

Thiele, A. A. (2014). Measuring Technical Debt.

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498–1516. http://doi.org/http://dx.doi.org/10.1016/j.jss.2012.12.052

Understand. (n.d.). Retrieved from https://scitools.com/

Vetro, A., Morisio, M., Torchiano, M., & Torino, P. (2008). An Empirical Validation of FindBugs Issues Related to Defects.

Villar, A., & Matalonga, S. (n.d.). Definiciones y tendencia de deuda técnica : Un mapeo sistemático de la literatura.

Visser, J. (2014). SIG / TÜViT Evaluation Criteria Trusted Product Maintainability, 1–19.

Wong, S., Cai, Y., Kim, M., & Dalton, M. (2011). Detecting Software Modularity Violations. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 411–420). New York, NY, USA: ACM. http://doi.org/10.1145/1985793.1985850

Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., & Shull, F. (2014). Comparing four approaches for technical debt identification, 403–426. http://doi.org/10.1007/s11219-013-9200-8