FACULTAD DE INGENIERÍA

CARRERA DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

“IMPACTO DEL USO DE HERRAMIENTAS DE SOFTWARE EN LA IMPLEMENTACIÓN DE SOFTWARE DE CALIDAD”

Tesis para optar el título profesional de:

Ingeniero de Sistemas Computacionales

Autor:

Dany Alexander Valdivia Huamán

Asesor:

M Sc. Ing. Laura Sofía Bazán Díaz

Cajamarca – Perú

2017

APROBACIÓN DE LA TESIS

La asesora y los miembros del jurado evaluador asignados, **APRUEBAN** la tesis desarrollada por el Bachiller **Dany Alexander Valdivia Huamán**, denominada:

“IMPACTO DEL USO DE HERRAMIENTAS DE SOFTWARE EN LA IMPLEMENTACIÓN DE SOFTWARE DE CALIDAD”

M Sc. Ing. Laura Sofía Bazán Díaz
ASESOR

Mg. Ing. Patricia Janet Uceda Martos
JURADO
PRESIDENTE

Mg. Ing. Christiaan Michael Romero Zegarra
JURADO

Mg. Ing. Yuri Alexis Túllume Mechán
JURADO

DEDICATORIA

A Dios, por haberme permitido llegar hasta este punto y haberme dado salud para lograr mis objetivos, además de su infinita bondad y amor.

A mis padres, por su apoyo incondicional, comprensión, cariño, por sus consejos, su ejemplo, por haberme formado con buenos valores y sentimientos.

A mis familiares, mis abuelos, mis tíos por apoyarme siempre en lo bueno y lo malo, a todos aquellos que participaron directa o indirectamente en la elaboración de esta tesis.

A todos, por esto y mucho más les dedico este trabajo de tesis, mi triunfo es el de ellos, gracias.

AGRADECIMIENTO

Me gustaría agradecer a Dios por bendecirme y permitirme llegar hasta este momento.

A mis padres por apoyarme y con su esfuerzo permitirme terminar mi carrera.

A la Universidad Privada del Norte por darme la oportunidad de estudiar y ser un gran profesional.

También me gustaría agradecer a mi asesora Mg. Sc. Ing. Laura Sofía Bazán Díaz quien durante todo el desarrollo de mi tesis, aportó con sus conocimientos, enseñanzas y por la amistad brindada.

A los docentes quien a lo largo de mi educación como profesional, me inculcaron conocimientos, valores y las ganas de ser cada día mejor.

Son muchas las personas que pasaron por mi vida y me ayudaron a salir adelante como profesional, a los cuales me encantaría agradecerles por su amistad, consejos, apoyo, ánimo y compañía en los momentos más difíciles de mi carrera.

Para ellos: muchas gracias y que Dios los bendiga.

ÍNDICE DE CONTENIDOS

Contenido

APROBACIÓN DE LA TESIS i

DEDICATORIA iii

AGRADECIMIENTO iv

ÍNDICE DE CONTENIDOS v

ÍNDICE DE TABLAS viii

ÍNDICE DE FIGURAS x

RESUMEN xii

ABSTRACT xiii

CAPÍTULO 1. INTRODUCCIÓN 1

CAPÍTULO 2. MARCO TEÓRICO 5

2.1. Antecedentes 5

2.2. Bases teóricas 7

2.2.1. Definición de herramienta software 7

2.2.2. Razones para el análisis de código 8

2.2.3. Ventajas y desventajas 8

2.2.4. Tipos de herramientas 9

2.2.5. Herramientas a usar 9

2.2.6. Clasificación según el tipo de análisis 12

2.2.7. Tipos de análisis 12

2.2.8. Ámbito de aplicación del análisis de código 12

2.2.9. Métricas a evaluar 13

2.2.10. Definición de software de calidad 14

2.2.11. Definición de calidad 14

2.2.12. Características de la calidad 14

2.2.13. Modelos de calidad 17

2.2.14. Métricas del producto 21

2.2.15. Cómo lograr la calidad del software 21
2.2.16. Factor de calidad elegido 22
2.2.17. Dimensión de las variables 22
2.3. Hipótesis 23
CAPÍTULO 3. METODOLOGÍA 24
3.1. Operacionalización de variables 24
3.2. Metodologías 25
3.2.1. Metodología de desarrollo de software 25
3.2.2. Desarrollo de la metodología del producto de aplicación profesional 29
3.2.3. Metodología de investigación científica 63
3.3. Nivel de investigación 63
3.4. Diseño de investigación 63
3.4.1. Grupos de estudio 64
3.5. Unidad de estudio 64
3.6. Población 64
3.7. Muestra 65
3.8. Técnicas, instrumentos y procedimientos de recolección de datos 65
3.8.1. Técnicas – Métodos 65
3.8.2. Instrumentos 65
3.9. Métodos, instrumentos y procedimientos de análisis de datos 65
3.9.1. Métodos 65
3.9.2. Procedimientos y herramientas 66
3.10. Viabilidad económica del proyecto....................66
3.10.1. Recursos humanos. 66
3.10.2. Materiales....................66
3.10.3. Técnicos....................67
3.10.4. Servicios....................67
3.10.5. Presupuesto 67
3.10.6. Financiamiento....................68
CAPÍTULO 4. RESULTADOS 69
CAPÍTULO 5. DISCUSIÓN 77
CAPÍTULO 6. CONCLUSIONES 79
CAPÍTULO 7. RECOMENDACIONES 81
CAPÍTULO 8. REFERENCIAS 82

ANEXOS 85

APÉNDICES 88

ÍNDICE DE TABLAS

Tabla n.° 1. Cuadro comparativo de herramientas software de calidad 9

Tabla n.° 2. Categorización de herramientas de calidad 11

Tabla n.° 3. Características de la calidad según la ISO 16

Tabla n.° 4. Modelo de calidad de McCall 17

Tabla n.° 5. Modelo de calidad de Dromey 18

Tabla n.° 6. Modelo de calidad FURPS 18

Tabla n.° 7. Operacionalización de variables 24

Tabla n.° 8. Tabla comparativa de metodologías de desarrollo de software 27

Tabla n.° 9. Lenguajes de programación soportados por SonarQube 35

Tabla n.° 10. Resultados de métricas de código. 43

Tabla n.° 11. Medición pre test, herramienta Designite 46

Tabla n.° 12. Medición pre test, herramienta SonarQube 47

Tabla n.° 13. Medición pre test, herramienta Code Metrics 47

Tabla n.° 14. Medición pre test, herramienta Cloc 48

Tabla n.° 15. Medición ciclo 1, herramienta Designite 48

Tabla n.° 16. Medición ciclo 1, herramienta SonarQube 49

Tabla n.° 17. Medición ciclo 1, herramienta Code Metrics 49

Tabla n.° 18. Medición ciclo 1, herramienta Cloc 50

Tabla n.° 19. Medición ciclo 2, herramienta Designite 50

Tabla n.° 20. Medición ciclo 2, herramienta SonarQube 51

Tabla n.° 21. Medición ciclo 2, herramienta Code Metrics 51

Tabla n.° 22. Medición ciclo 2, herramienta Cloc 52

Tabla n.° 23. Medición ciclo 3, herramienta Designite 52

Tabla n.° 24. Medición ciclo 3, herramienta SonarQube 53

Tabla n.° 25. Medición ciclo 3, herramienta Code Metrics 53

Tabla n.° 26. Medición ciclo 3, herramienta Cloc.....54
Tabla n.° 27. Medición ciclo 4, herramienta Designite.....54
Tabla n.° 28. Medición ciclo 4, herramienta SonarQube.....55
Tabla n.° 29. Medición ciclo 4, herramienta Code Metrics.....55
Tabla n.° 30. Medición ciclo 4, herramienta Cloc.....56
Tabla n.° 31. Medición ciclo 5, herramienta Designite.....56
Tabla n.° 32. Medición ciclo 5, herramienta SonarQube.....57
Tabla n.° 33. Medición ciclo 5, herramienta Code Metrics.....57
Tabla n.° 34. Medición ciclo 5, herramienta Cloc.....58
Tabla n.° 35. Medición post test, herramienta Designite.....58
Tabla n.° 36. Medición post test, herramienta SonarQube.....59
Tabla n.° 37. Medición post test, herramienta Code Metrics.....59
Tabla n.° 38. Medición post test, herramienta Cloc.....60
Tabla n.° 39. Compacto de resultados obtenidos en los ciclos de medición.....61
Tabla n.° 40. Diseño experimental.....63
Tabla n.° 41. Subdivisión de grupos de estudio.....64
Tabla n.° 42. Total de líneas de código.....65
Tabla n.° 43. Presupuesto del proyecto.....67
Tabla n.° 44. Estadísticos de prueba SonarQube.....69
Tabla n.° 45. Estadísticos de prueba Code Analysis.....69
Tabla n.° 46. Estadísticos de prueba Cloc.....70
Tabla n.° 47. Niveles de impacto.....70
Tabla n.° 48. Cálculo del impacto de los indicadores.....71

ÍNDICE DE FIGURAS

Figura n.° 1. Modelo ISO 2196 20
Figura n.° 2. Modelo de calidad de Boehm 20
Figura n.° 3. Metodología Scrum 26
Figura n.° 4. Tablero de Kanban 26
Figura n.° 5. Metodología XP 27
Figura n.° 6. Dashboard de JIRA Software 31
Figura n.° 7. Tablero Kanban para el proyecto SISGED 32
Figura n.° 8. Pantalla de configuración de WIP en tablero Kanban 32
Figura n.° 9. Visualización de tarjetas Kanban en JIRA 33
Figura n.° 10. Tarjeta Kanban en JIRA 34
Figura n.° 11. Configuración de proyecto en SonarQube 37
Figura n.° 12. Administración de proyectos en SonarQube 37
Figura n.° 13. Salida consola análisis con SonarQube Scanner for MSBuild 39
Figura n.° 14. Dashboard SonarQube 39
Figura n.° 15. Indicadores proyecto SISGED 40
Figura n.° 16. Designite 41
Figura n.° 17. Resumen de análisis Designite 42
Figura n.° 18. Resultados Visual Studio Code Metrics 44
Figura n.° 19. Resultados de Visual Studio Code Metrics por proyecto y clase 45
Figura n.° 20. Resultados Cloc 46
Figura n.° 21. Diseño quasi-experimental 63
Figura n.° 22. Velocidad de carga en pre test 72
Figura n.° 23. Velocidad de carga en post test 73
Figura n.° 24. Definición de métricas de Mantenibilidad de SonarQube 73
Figura n.° 25. Evolución de cobertura de código 74
Figura n.° 26. Evolución de líneas duplicadas 75

Figura n.° 27. Evolución de bloques duplicados 75

Figura n.° 28. Evolución de complejidad ciclomática 76

Figura n.° 29. Evolución de líneas de código 76

Figura n.° 30. Instalación de Designite 90

RESUMEN

Perú es uno de los países latinoamericanos que no tuvo mucho éxito en levantar una industria de software de calidad en comparación con los otros países de la región y mucho menos con países líderes en software. Esta falta de éxito se debe a la poca importancia que se le viene dando a la calidad de software, principalmente por tema de costos y además por la no existencia de profesionales conocedores de herramientas o tecnologías que agilicen esta labor.

En la presente tesis, se determina el impacto del uso de herramientas de calidad en la implementación de software de calidad. Asimismo se hace énfasis en la necesidad del uso de herramientas de calidad por parte de profesionales y empresas dedicadas al desarrollo de software que deseen asegurar la entrega de software de calidad.

Las herramientas de calidad empleadas fueron SonarQube, Visual Studio Code Metrics, Designite y Cloc, las cuales se usaron para medir indicadores de calidad en el código fuente del sistema SISGED. Entre los principales indicadores de calidad abarcados tenemos: porcentaje de código repetido, complejidad ciclomática, cobertura de código, deuda técnica y total de líneas de código; dichas herramientas nos muestran posibles soluciones a los problemas de código encontrados en el sistema SISGED, estas se aplicaron a lo largo del desarrollo y se analizó cómo iba evolucionando la calidad interna y externa del sistema.

Como resultado final, se obtuvo que es poco probable medir el porcentaje de mejora total de un sistema evaluado con herramientas de calidad, ya que la medición de los indicadores se realiza en distintas escalas y estos se agrupan de diferente manera; mas sí se pudo determinar que el uso de herramientas de calidad producen un impacto positivo en la implementación de software de calidad.

ABSTRACT

Peru is one of the Latin American countries that has not been very successful in building a quality software industry compared to other countries in the region, much less with leading countries in software. This lack of success is due to the lack of importance given to the quality of software, this mainly because of costs and also because of the lack of professionals who know the tools or technologies to speed up this work.

In this thesis, the impact of the use of quality tools in the implementation of quality software is determined. Also emphasizes the need for the use of quality tools by professionals and companies dedicated to the development of software that wish to ensure the delivery of quality software.

The quality tools used were SonarQube, Visual Studio Code Metrics, Designite and Cloc, which were used to measure quality indicators in the source code of the SISGED system. Among the main quality indicators covered are: percentage of repeated code, cyclomatic complexity, code coverage, technical debt and total of lines of code; these tools show possible solutions to the code problems found in the SISGED system, which were applied throughout the development and analyzed how the internal and external quality of the system was evolving.

As a final result it was obtained that it is unlikely to measure the percentage of total improvement of a system that has been evaluated with quality tools, since the measurement of the indicators is realized in different measures and these are grouped of different way; But it was possible to determine that the use of quality tools has a positive impact on the implementation of quality software.

CAPÍTULO 1. INTRODUCCIÓN

1.1. Realidad problemática

(Robert, 2016), no presenta la reputación que ha logrado el desarrollo de software hoy en día, para lo cual sostuvo que:

> Por un lado, permite crear productos software "mágicos" de capacidad casi ilimitada, que pueden superar y sustituir a los humanos hasta tal punto de dejarnos sin trabajo en el futuro. Pero, simultáneamente a muchos de estos productos de software se les acusa de ser difíciles de usar y que fallan muy a menudo. Las interrogantes frecuentes son: ¿Por qué hay tantas quejas sobre la calidad del software?, ¿Acaso en 2017 aún no hemos aprendido a desarrollar software?, ¿Es tan complejo el proceso de desarrollo?. En relación a la calidad del software hay dos cuestiones complementarias que hay que resolver:

- La primera cuestión es la verificación, que consiste en asegurar que el software esté libre de errores. Esta comprobación puede realizarse mediante pruebas de diferentes clases o bien mediante herramientas que analizan el código para encontrar errores.
- La segunda cuestión es la validación, estudiar si el software satisface los requisitos y expectativas de todos los stakeholders. Los requisitos pueden ser de tipo funcional ("la aplicación debe hacer X") o de tipo no funcional ("la aplicación debe funcionar aunque no haya conexión a Internet").

No debería ser una sorpresa que la calidad de software es una de las principales preocupaciones para las organizaciones de todos los tamaños y formas. Pero nos preguntamos si estas organizaciones cumplen con los principales estándares de calidad o cómo mantienen la calidad de su software. En un estudio realizado entre 600 profesionales de desarrollo de software, testers y profesionales de TI y operaciones, representando a más de 30 industrias diferentes, se concluyó que: a) El 67% de profesionales están de acuerdo con la calidad del software que construyen, mientras que sólo un 14% están muy de acuerdo y un pequeño 14% consideró negativa la calidad del software que construyen; esto nos hace notar que hay bastantes mejoras por hacer en cuanto a la calidad del código, ya que un tercio de los participantes no estaban convencidos de decir que se sentían seguros con la calidad del software que construyen, b) Cuando se trata de la calidad de código, los participantes estuvieron de acuerdo en que la revisión de código es el método número uno para la mejora de la calidad del código, ya sea revisión manual o mediante alguna herramienta de análisis estático de código y está claro que los equipos entienden su importancia ahora más que antes, c) En cuanto a los mayores beneficios de hacer

revisiones de código se encontró lo siguiente: un 90% dijo que mejoran la calidad del código, un 72% dijo que aumenta el intercambio de conocimiento a través del equipo, un 59% dijo que aumenta la mantenibilidad de código y un 56% dijo que hace fácil el aprendizaje de los desarrolladores con menos experiencia. Sin embargo, lo más importante a resaltar es que los equipos de desarrollo están haciendo revisiones de código basado en herramientas más a menudo (Pinkham, 2016).

La calidad del software es muy importante en el entorno competitivo de hoy en día. Es una restricción crítica en los proyectos de software. Los principales objetivos de las organizaciones de software son entregar los productos a tiempo y lograr objetivos de calidad. La calidad depende directamente de los procesos de software, que son intrínsecamente variables e inciertos, con riesgos sustanciales. La gestión del riesgo de calidad es un desafío importante. El enfoque convencional de la gestión del riesgo de calidad para los procesos de software en curso tiene dos diferencias importantes: se utilizan los modelos analíticos estáticos y no se aplican sistemáticamente metodologías estructuradas para mejorar los procesos y mejorar la calidad (Bubevski, 2013).

La (Universidad La Salle, 2015), sostuvo que:

> Perú intentó por años levantar la industria de software sin tener mucho éxito. Existen varias empresas de software muy pequeñas, comparadas con otros países de la región, ni que decir de otros países líderes en software; un resultado bien pobre, considerando que existen más de cien carreras de informática/sistemas en todo el Perú y cientos de institutos de informática que enseñan sintaxis de algunos lenguajes de programación y una que otra tecnología usada en el medio. Si bien, decenas de miles de personas son formadas bajo este modelo; entonces, ¿cómo es posible que nuestra industria sea tan pequeña con tantos profesionales?. El problema es que se le está dando más importancia al levantamiento de requerimientos o proceso de diseño que al proceso de construcción del software en sí mismo. La industria de software a nivel mundial progresa bastante en este sentido y aplica la analogía de esta forma: el ingeniero sigue siendo aquel que diseña software, que puede empezar usando lenguajes de modelamiento para especificar los requerimientos del cliente o problema a resolver; pero este proceso de diseño ejecutado por el ingeniero no acaba en documentos que listan las especificaciones o en un documento de diseño de las partes del software. El ingeniero tiene la gran responsabilidad de escribir el diseño final e indiscutible: código fuente de calidad. Pero adicional a esto, el ingeniero tiene el reto de ganar más conocimientos en escalabilidad, mantenimiento de software, pruebas, lidiar con código pre-existente y aprender a trabajar en equipo.

Roberto (2010), nos presenta una lista con los principales problemas causados por la mala calidad del software:

- Las organizaciones no saben cómo garantizar la calidad del software que producen.
- Desconocen las herramientas para analizar la calidad del código.
- Desconocen sobre las métricas de calidad de software.
- Importantes organizaciones se centran en invertir en la mejora de calidad del software, pero la mayoría desconoce cómo mejorar los productos que ofrecen.
- Los pequeños fallos en el software provocan una gran inversión en buscar su solución.
- Los costos de mantenimiento son grandes.

1.2. Formulación del problema

¿En qué medida impacta el uso de herramientas que miden la calidad de código en la implementación de software de calidad?

1.3. Justificación

En el presente estudio se busca determinar el nivel de impacto que tiene el uso de herramientas software de análisis y revisión de código en el aseguramiento e implementación de productos software de calidad. Algunos estudios indican que las revisiones de código son una de las mejores formas de garantizar e implementar software de calidad y el beneficio que se obtiene de hacer revisiones de código basadas en herramientas, es la influencia positiva sobre la calidad de software que se ayuda a construir (Pinkham, 2016).

Por otro lado, resulta evidente que la situación local y nacional de la calidad de software no es la adecuada si se compara con países líderes en desarrollo de software, dicho esto, la investigación será de gran aporte si se demuestra que el impacto del uso de herramientas software en la implementación de software de calidad es considerablemente positivo. De ser el caso que el impacto resulta positivo, todo aquel que desarrolla software puede llegar a utilizar las herramientas mencionadas en el estudio independientemente del lenguaje utilizado en su organización.

Adicionalmente, la presente investigación se realiza para poner en práctica todos los conocimientos adquiridos durante los cinco años de carrera en la Universidad Privada del Norte; producto de esto será la obtención del título profesional en la carrera de Ingeniería de

Sistemas Computacionales; además significa un aumento en el acervo bibliográfico sobre un nuevo tema que es la mejora de calidad de software.

Finalmente, los resultados y herramientas a utilizar podrían servir como base, primero para futuras investigaciones sobre calidad de software y herramientas que aseguran la calidad, por otro lado, para mejorar la industria local y nacional de desarrollo de software.

1.4. Limitaciones

Entre las principales dificultades encontradas para el desarrollo de la presente investigación, tenemos:

- La falta de investigaciones locales y nacionales, relacionadas con el tema a tratar, así como la falta de bibliografía local.
- La mayoría del material bibliográfico encontrado se encuentra en idioma inglés a nivel avanzado.
- La poca importancia que se le da actualmente al desarrollo de software y la escasez de empresas desarrolladoras de software en la localidad de Cajamarca.

1.5. Objetivos

1.5.1. Objetivo general

- Identificar el impacto de las herramientas de calidad de código en la implementación de un producto software de calidad.

1.5.2. Objetivos específicos

- Describir las principales herramientas que aseguran la implementación de software de calidad.
- Calcular el porcentaje de mejora de un producto software con la utilización de herramientas de calidad de código.
- Aplicar las mejoras propuestas por las herramientas de calidad utilizando la metodología de desarrollo de software Kanban.
- Determinar la influencia de la calidad interna del código sobre la calidad externa de un producto software.
- Medir el nivel de mantenibilidad de un producto software analizado con herramientas de calidad.
- Medir el nivel de mejora de código con el uso de herramientas de calidad.

CAPÍTULO 2. MARCO TEÓRICO

2.1. Antecedentes

Al hablar de la situación actual del desarrollo de software, Wailgun (2010), sostuvo que:

> Se caracteriza por la escasez de tiempo y de personal, así como intensas presiones de costos, causando que "la necesidad de velocidad" sea lo más importante para los equipos de desarrollo; en lugar de asegurarse que la calidad sea el trabajo n° 1. Pero esto no significa que los equipos de desarrollo de aplicaciones tengan que aceptar las cosas tal como son ("el software siempre tendrá errores") y desechar cualquier idea de mejorar la calidad de su código (p. 20).
>
> El autor indica que "la calidad no puede ser rociada en una aplicación justo antes de ser expuesta a sus clientes", más bien debe ser una parte del ciclo completo de vida del desarrollo de software, desde el inicio de su implementación. El autor concluye indicando que el software debe ser diseñado de modo que sea más limpio, fácil de evaluar y volver a trabajar, lo cual significa que el código tendrá menos errores y que los bugs serán más fáciles de diagnosticar y reparar.

En un segundo trabajo, tenemos a Martin (2011), presentando en su libro un variado número de técnicas aprendidas durante su experiencia como desarrollador, para la correcta construcción de un producto software de calidad. En este trabajo se presentan buenas prácticas de programación como: correcta nomenclatura de nombres y variables, correcto uso de funciones, máxima cantidad de líneas permitidas por función o método, no duplicidad de código, comentarios de calidad, cobertura de pruebas unitarias, entre otras. Esto aporta a la investigación ya que las herramientas a utilizar calculan estas métricas de calidad en base al análisis estático del código. El autor nos garantiza que de seguir estas técnicas propuestas se tendrá asegurado el desarrollo de un software de calidad y desde el punto de vista de las organizaciones una reducción considerable en los costos de mantenimiento.

En un tercer trabajo, tenemos a Pressman (2010), quien nos presenta el control y aseguramiento de la calidad como actividades esenciales para cualquier negocio que genere productos que utilicen otras personas. Este trabajo nos resume cómo va evolucionando el control de calidad de los productos y/o servicios y cómo surge el aseguramiento de la calidad en el desarrollo de software, el cual inicialmente sólo era responsabilidad única del programador. Un aspecto importante a tener en cuenta en este trabajo es el uso de métricas para medir la calidad del código, definida en lo siguiente: el código fuente y los productos del trabajo relacionados deben apegarse a los estándares

locales de codificación y tener características que faciliten darle mantenimiento. Aquí hay dos cosas importantes que considerar, métricas y código fuente; para lograr tener calidad en el código fuente se debe revisar que se utilicen patrones de diseño, que el software sea fácil de mantener, se debe respetar el porcentaje de comentarios internos y que la complejidad ciclomática se encuentre en el rango permitido. Esto aporta a la presente investigación, ya que las herramientas a utilizarse se encargan de analizar las métricas mencionadas anteriormente. El autor concluye indicando que la única forma de lograr una ingeniería de software madura es realizando el análisis de estas métricas durante el proceso de desarrollo del producto software.

Otro trabajo corresponde a Sharma et al. (2016), quienes presentan la herramienta Designite, como una herramienta de evaluación de la calidad de diseño de software y que además esta herramienta no sólo es compatible con el diseño integral de la detección de code smells, sino que también proporcionan un análisis detallado de métricas. También ofrece varias características para ayudar a identificar problemas que contribuyen a diseñar la deuda y mejorar la calidad del diseño del sistema de software de análisis. Los autores concluyen que la herramienta Designite es eficaz en la evaluación de calidad de diseño de software, soportar la detección completa de olores de diseño, análisis detallado de métricas de código y además se integra con herramientas como SonarQube. Con todas estas características, Designite evalúa la calidad de diseño de un sistema de software y ayuda a mejorar la agilidad del diseño del software.

También, tenemos a Fontana et al. (2016), quienes nos presentan cinco herramientas que proporcionan algún tipo de índice que sirve para evaluar la calidad de un proyecto software, el cual se utilizará al gestionar su deuda técnica. Una de las herramientas presentadas como plataforma para gestionar la calidad del código es SonarQube, que se encarga de obtener todas las violaciones de reglas de código, estas violaciones son tomadas cómo un índice de deuda técnica para un posterior cálculo del costo de desarrollo estimado que tomaría reescribir el proyecto desde cero. El antecedente contribuye así a reforzar la hipótesis de que el uso de estas herramientas tiene un impacto positivo en la mejora de la calidad de un producto software y adicionalmente la reducción de la deuda técnica del producto final, que básicamente son las consecuencias de un desarrollo apresurado que puede resultar en errores por no aplicarse patrones ni buenas prácticas de programación.

Toshisa et al. (2002), nos hablan claramente del objetivo de su investigación, recalcando que es muy importante para mejorar la calidad del software, usar programas de análisis, herramientas de medición y métodos de aseguramiento de calidad de software en los puntos apropiados durante el proceso de desarrollo, ya que cómo indican los autores, en

muchos departamentos de desarrollo, a menudo no hay tiempo suficiente para evaluar y utilizar las herramientas y métodos de aseguramiento de calidad de software. Los autores se centran en un punto importante a considerar que es la utilización de herramientas de análisis estático de código para detectar componentes del software propenso a fallos; y concluyen que la calidad del software se logra aumentando la eficiencia de las revisiones o pruebas con la ayuda de estas herramientas.

Además tenemos a Seymor (2014), quien nos presenta la herramienta Coverity, que combina el análisis de código y optimiza la eficiencia de la detección de defectos en bases de código c# para encontrar problemas que podrían dar lugar a futuros accidentes, además de permitir a las organizaciones de desarrollo crear y entregar un mejor software y más rápido; un punto muy importante, es que el autor concluye que las organizaciones deben integrar las pruebas más temprano en el proceso de desarrollo para ayudar a garantizar la alta calidad y la seguridad de su código fuente; también agregó que las características ofrecidas por este tipo de herramientas de calidad pueden ayudar a los equipos de desarrollo de software a mejorar y liberar más rápido y eficientemente sus productos software.

Como última investigación tenemos a Pinkham (2016), quien en uno de sus últimos estudios realizados y con participación de 600 desarrolladores, testers, profesionales de TI y operaciones de todo el mundo concluye que las revisiones de código son una de las mejores de formas de garantizar e implementar software de calidad y el beneficio que se obtiene de hacer revisiones de código basadas en herramientas, es la influencia positiva sobre la calidad de software que se ayuda a construir.

2.2. Bases teóricas

2.2.1. Definición de herramienta software

Un programa que se emplea en el desarrollo, reparación o mejora de otros programas o de hardware. Tradicionalmente, un conjunto de herramientas de software dirigido a las necesidades esenciales durante el desarrollo del programa: un conjunto típico podría consistir en un editor de texto o un compilador, y algún tipo de herramienta de depuración. Dicho conjunto se concentra únicamente en la fase de producción de programas y que es normalmente proporcionado por un sistema de desarrollo del programa (Oxford University Press, 2004).

Herramientas de calidad del software, son herramientas que realizan un control desde el punto de vista del estudio estático y de caja blanca, es decir, analizan las fuentes del software sin la necesidad de ejecutarlo (Garzas, 2012).

2.2.2. Razones para el análisis de código

Roberto (2010), nos detalla una lista de grandes razones para el análisis de código fuente:

- Es un medio que nos permite mejorar, no es un fin en sí mismo.
- Permite validar las reglas metodológicas aplicadas en el proyecto.
- La detección de incidencias, permiten la adecuada localización de errores que pasan desapercibidos en el ciclo de desarrollo.
- Partiendo de una metodología adecuada y usando el análisis de código como un apoyo a la calidad.
- El análisis de código, facilita con una mínima inversión de tiempo, la localización de defectos permitiendo un alto grado de retorno de inversión.
- El análisis manual, por su coste deberá ser abordado en fases criticas de proyectos de desarrollo de software o en proyectos críticos.
- El análisis automático, puede ser realizado con una mayor periodicidad ya que no requiere de intervención y puede ser programado y repetido tantas veces como sea necesario, dotando el proyecto de un mecanismo ágil de validación de certificación.

2.2.3. Ventajas y desventajas

Basado en el documento de OWASP donde se especifican las ventajas y desventajas del análisis de código estático, se puede enumerar los siguientes puntos (Ospina, 2015).

- Ventajas
  1) Escala bien, puede ser implementado en múltiples proyectos y es repetible.
  2) Pueden detectar con alta confianza vulnerabilidades bien documentadas como los desbordamientos de pila, inyecciones SQL, etc.
  3) La ejecución de análisis generalmente es rápida.
  4) Detectan la causa del problema, pues se realiza una búsqueda directa en el código, mientras que los test de penetración solo establecen el problema, más no el motivo.

- Desventajas
  1) Muchas clases de vulnerabilidades son difíciles de encontrar automáticamente, como los problemas de autenticación, los accesos de control, la lógica de negocio,

etc. El estado del arte actual solo permite encontrar un pequeño porcentaje de las vulnerabilidades.

2) Los resultados contiene un alto número de falsos positivos.
3) Generalmente no pueden encontrar problemas de configuración, pues estos archivos no están representados en el código.
4) Muchas de estas herramientas tiene dificultades para analizar código que no se encuentra compilado.

2.2.4. Tipos de herramientas

Javier Garzas (2012), nos presenta un listado con las principales herramientas software que nos facilitan y aseguran entregar un software de calidad, las cuales se están agrupando por funcionalidad:

1) Herramientas de calidad del software.
    - SonarQube
    - Google CodePro Analytix
    - Simian
    - Cloc
    - Visual Studio Code Metrics
    - Designite
    - Coverity
2) Herramientas de Testing.
    - Selenium
    - JMeter
    - Testlink
3) Herramientas para Scrum.
    - JIRA Software
    - ScrumDo
    - IceScrum

2.2.5. Herramientas a usar

2.2.5.1. Herramientas disponibles

La tabla n.° 1 muestra un cuadro comparativo con las herramientas software de calidad más conocidas en el mercado.

Tabla n.° 1. Cuadro comparativo de herramientas software de calidad

	Características	Lenguajes soportados	Desventajas
SonarQube	Se puede instalar en un equipo local o remoto. Tiene una versión para almacenar los resultados en la nube.	Multilenguaje. Más de 20 lenguajes de programación soportados, incluyendo Java, C/C++, C#, HTML, etc.	Algunos plugins son pagados.
Code Metrics	Viene incluido en Visual Studio. El análisis se puede realizar desde Visual Studio y la consola de Windows.	C/C++ C#	Solo analiza proyectos realizados en la plataforma .NET
Designite	Muestra un amplio detalle de las métricas evaluadas. El análisis realizado es más extenso, por ejemplo detecta olores de arquitectura y diseño.	C/C++ C#	Solo analiza proyectos realizados en la plataforma .NET
Cloc	No requiere instalación y es súper fácil de usar. Reconoce el número de líneas en blanco dentro del código, y aquellas que corresponden a texto de documentación y comentarios.	Multilenguaje, incluyendo C#, ASP.NET, VB.NET y muchos más.	Solo se centra en una característica: contar líneas de código.
Coverity	Detecta fallas de software y vulnerabilidades de seguridad.	Android, Javascript, Ruby, C/C++/Objective C, C#, Python, Java	Va más por el lado de detectar vulnerabilidades de seguridad de un software.
Kiuwan	Detecta defectos en el código fuente.	Multilenguaje, incluyendo C#, Java, Javascript y muchos más.	Herramienta en Cloud que implica estar conectado a Internet.
Simian	Identifica duplicados en archivos de código fuente e incluso archivos de texto sin formato.	Java, C#, C, C++.	Al igual que Cloc, solo se centra en una característica: identificar

	Características	Lenguajes soportados	Desventajas
			duplicados.
PMD	Identifica variables no usadas, bloques vacíos, creación de objetos innecesaria.	Java, Javascript, XML.	Trabaja principalmente con lenguajes Java.

Fuente: Elaboración propia

2.2.5.2. Elección de herramientas

A continuación se presenta las herramientas software de calidad que se utilizaron en la presente investigación y el motivo por el cual se las eligió, pero antes vamos a realizar una categorización de todas las herramientas presentadas. Para dicha categorización se eligieron factores mencionados en el marco teórico y que se relacionan con las dimensiones de la variable herramienta de calidad. Estos factores se puntuarán en una escala el 0 al 2, donde: 0 significa bajo o escaso, 1 medio y 2 alto.

Tabla n.° 2. Categorización de herramientas de calidad

Herramienta	Nivel de aprendizaje	Facilidad de uso	Documentación/Soporte	Precio	Puntuación total
SonarQube	2	2	2	2	8
Code Metrics	2	2	1	2	7
Designite	1	2	1	1	5
Cloc	1	2	1	2	6
Coverity	0	2	2	0	4
Kiuwan	1	1	2	0	4
Simian	0	1	2	0	4
PMD	1	1	1	1	4

Fuente: Elaboración propia

Por lo tanto, con las puntuaciones finales calculadas, podemos observar que cuatro herramientas sobresalen de las demás, esto debido a que son libres de pago, cuentan con una documentación actualizada y sobre todo son fáciles y rápidas de aprender, además tenemos que:

- SonarQube: cuenta con un cuadro de mando con información entendible.
- Cloc: no requiere instalación.
- Code Metrics: viene integrada en el IDE Visual Studio y se puede programar la ejecución en cada compilación del proyecto.

- Designite: muestra un Dashboard bien detallado de olores de código existentes en la solución.

2.2.6. Clasificación según el tipo de análisis

Estas herramientas de mejora de calidad de software se clasifican de acuerdo al punto específico que evalúan e intentan mejorar, entre las categorías más importantes tenemos (Cruz, 2010):

- Análisis de código: herramientas que analizan y depuran el código, informando al desarrollador problemas de olores de código, como: nombres de variables sin sintaxis, métodos sin usar, duplicidad de código, métodos complejos, etc.
- Análisis de estilo: estas herramientas analizan el código del programa, buscando que cumpla con una serie de reglas de estilo y buenas prácticas de programación, como: revisión de cabeceras, importación de paquetes, declaración de atributos y variables.
- Análisis de documentación: estas herramientas se especializan en analizar la documentación que acompaña el código.

2.2.7. Tipos de análisis

Roberto (2010), establece dos tipos de análisis que se pueden realizar al código fuente:

- Análisis estático: el análisis estático de código consiste en el análisis de un sistema informático mediante la inspección directa de la fuente u objeto codificado que describe el sistema con respecto a la semántica del código. La experiencia demuestra que muchos de los planteamientos que la industria considera demasiado costoso (como la especificación formal y el análisis estático de código) en realidad pueden reducir y reducen el costo total.

- Análisis dinámico: consiste en el análisis de las aplicaciones informáticas, mediante la ejecución de los programas en un procesador real o virtual. Para que el análisis dinámico pueda ser eficaz, deberá ser ejecutado con un conjunto de entradas de prueba suficientes para producir comportamiento interesante. Además permite la obtención de métricas tales, como cobertura de código, niveles de ejecución, etc.

2.2.8. Ámbito de aplicación del análisis de código

De igual manera, Roberto (2010) nos presenta las posibles áreas de análisis para la revisión manual de código:

- Aspectos del diseño de una API.

- Aspectos de la arquitectura (como la elección de enlaces de cliente y la herencia).
- Elección de estructuras de datos y algoritmos.
- Estilos de programación.
- Comentarios y documentación.
- La adherencia a prácticas de codificación oficial.
- Los desarrolladores deben discutir si una cierta clase está realmente justificada o debe pasar sus funcionalidades a otra, o, por el contrario, si una potencial clase se perdió.
- El diseño de las API o interfaces es esencial para su uso por otros elementos de software y permitir su reutilización.

2.2.9. Métricas a evaluar

Entre las métricas elegidas para evaluar la calidad del software, tenemos:

- Porcentaje de código repetido: según Martin (1999), esta métrica es el peor enemigo de la mantenibilidad y que aumenta innecesariamente el número de líneas de código, dispara los costes y aumenta los riesgos.
- Complejidad ciclomática: es una métrica que proporciona una medición de la complejidad lógica de nuestro código, está basada en la teoría de grafos. Es una de las métricas de software de mayor aceptación, ya que es concebida para ser independiente del lenguaje (Aldazabal Gil, 2015).
- Cobertura de código: según la Microsoft Developer Network, se usa para determinar qué proporción de código del proyecto se está probando realmente mediante pruebas codificadas como pruebas unitarias.
- Deuda técnica: según Javier Garzas (javiergarzas.com, 2012), es el coste y los intereses a pagar por hacer mal las cosas. El sobre esfuerzo a pagar para mantener un producto software mal hecho, y lo que conlleva, como el coste de la mala imagen frente a los clientes.
- Líneas de código: el Centro Tecnológico de Calidad de Software de la NASA, explica que el tamaño es una de las formas más antiguas y comunes de medición de software. El tamaño de los módulos es en sí misma un indicador de calidad. El tamaño puede ser medido a través del total de líneas de código, contando todas las líneas; que no esté en blanco, disminuyendo las líneas totales, por el número de espacios en blanco y comentarios, y todas aquellas sentencias ejecutables que se definen por un delimitador dependiente del lenguaje de programación.

2.2.10. Definición de software de calidad

El software que cumple con los requerimientos del negocio, proporciona al usuario una experiencia gratificante, y tiene menos defectos (Wailgum, 2010).

Proceso eficaz de software que se centra en crear un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan (Pressman, 2010).

Según la definición anterior, se puede enfatizar tres puntos importantes:

- Un proceso eficaz de software establece la infraestructura que da apoyo a cualquier esfuerzo de elaboración de un producto de software de alta calidad.
- Un producto útil entrega contenido, funciones y características que el usuario final desea.
- Al agregar valor para el productor y para el usuario de un producto, el software de alta calidad proporciona beneficios a la organización que lo produce y a la comunidad de usuarios finales.

2.2.11. Definición de calidad

Entre las principales definiciones de calidad, tenemos:

- La calidad es la suma de todos aquellos aspectos o características de un producto o servicio que influyen en su capacidad de satisfacer las necesidades explicitas o implícitas (International Organization for Standardization, 1995).
- Es el grado con el cual el cliente o usuario percibe que el software satisface sus expectativas (IEEE, 2010).
- Grado en el que un conjunto de características inherentes cumple con los requisitos (International Organization for Standardization, 2015).

2.2.12. Características de la calidad

La Organización Internacional de Estandarización (ISO) provee un estándar para la evaluación de calidad de software (International Organization for Standardization, 2015), donde se define seis características de un producto software de calidad. Estas características son:

- Funcionalidad: indica en qué medida las funciones específicas están disponibles en el software.
- Confiabilidad: indica la confiabilidad del software.

- Usabilidad: indica en qué medida la interfaz de usuario es fácil de usar para el usuario.
- Eficiencia: indica la capacidad del software de cumplir con sus requerimientos internos y externos.
- Mantenibilidad: indica en qué medida el software puede ser fácilmente construido y mantenido.
- Portabilidad: indica la facilidad de adaptación del software de un ambiente a otro.

En la tabla n.° 3. Se observa los indicadores relacionados a cada característica de la calidad según la ISO.

Tabla n.° 3. Características de la calidad según la ISO

Funcionalidad	Fiabilidad	Usabilidad	Eficiencia	Mantenibilidad	Portabilidad
• Aptitud • Precisión • Interoperabilidad • Conformidad • Seguridad • Trazabilidad	• Madurez • Tolerancia a fallas • Recuperabilidad • Disponibilidad • Degradabilidad	• Comprensibilidad • Facilidad de aprendizaje • Operatividad • Explicitud • Adaptabilidad al usuario • Atractividad • Claridad • Facilidad de ayudas • Amistoso al usuario	• Respecto al tiempo • Respecto a los recursos	• Facilidad de análisis • Facilidad de cambio • Estabilidad • Facilidad de prueba	• Adaptabilidad • Facilidad de instalación • Conformidad • Reemplazabilidad

Fuente: (International Organization for Standardization, 2015)

2.2.13. Modelos de calidad

A continuación, se presenta los modelos de calidad del software y cada uno de los factores o métricas de calidad asociados:

2.2.13.1. Modelo McCall

El modelo de McCall describe la calidad como un concepto elaborado mediante relaciones jerárquicas entre factores de calidad, en base a criterios y métricas de calidad (Dirección General de Servicio Civil, 2013).

Este enfoque es sistemático y permite cuantificar la calidad a través de las siguientes fases:

- Determinación de los factores que influyen sobre la calidad del software.
- Identificación de los criterios para juzgar cada factor.
- Definición de las métricas de los criterios y establecimiento de una función de normalización que define la relación entre las métricas de cada criterio y los factores correspondientes.
- Evaluación de las métricas.
- Correlación de las métricas a un conjunto de guías que cualquier equipo de desarrollo podría seguir para la colección de métricas.

La tabla n.° 4. Muestra los factores de calidad y sus criterios asociados.

Tabla n.° 4. Modelo de calidad de McCall

Puntos de vista	Factor	Criterios
Operación	• Corrección • Fiabilidad • Eficiencia • Integridad • Facilidad de uso	• ¿Hace el software lo que yo quiero? • ¿Lo hace de forma exacta todo el tiempo? • ¿Se ejecuta sobre mi Hardware lo mejor posible? • ¿Es seguro? • ¿Puedo ejecutarlo?
Revisión	• Facilidad de mantenimiento • Facilidad de prueba • Flexibilidad	• ¿Puedo arreglarlo? • ¿Puedo probarlo? • ¿Puedo modificarlo?
Transición	• Interoperabilidad • Portabilidad • Reusabilidad	• ¿Podré comunicarlo con otros sistemas? • ¿Podré utilizarlo en otra máquina? • ¿Podré reutilizar parte del software?

Fuente: (Dirección General de Servicio Civil, 2013)

2.2.13.2. Modelo de Dromey

Dromey propuso un marco de referencia para construcción de modelos de calidad, basado en como las propiedades medibles de un producto de software pueden afectar los atributos de calidad generales, como por ejemplo, confiabilidad y mantenibilidad (Dirección General de Servicio Civil, 2013).

La tabla n.° 5. Presenta la relación que establece Dromey entre las propiedades de calidad del producto y los atributos de calidad de alto nivel.

Tabla n.° 5. Modelo de calidad de Dromey

Factor	Criterio
Correctitud	• Funcionalidad • Confiabilidad
Internas	• Mantenibilidad • Eficiencia • Confiabilidad
Contextuales	• Mantenibilidad • Reusabilidad • Portabilidad • Confiabilidad
Descriptivas	• Mantenibilidad • Reusabilidad • Portabilidad • Usabilidad

Fuente: (Dirección General de Servicio Civil, 2013)

2.2.13.3. Modelo FURPS

El modelo de McCall sirvió de base para modelos de calidad posteriores, y este es el caso del modelo FURPS. En este modelo se desarrollan un conjunto de factores de calidad de software, bajo el acrónimo de FURPS: funcionalidad (Functionality), usabilidad (Usability), confiabilidad (Reliability), desempeño (Performance) y capacidad de soporte (Supportability). La tabla n.° 6 presenta la clasificación de los atributos de calidad que se incluyen en el modelo, junto con las características asociadas a cada uno (Pressman, 2010).

Tabla n.° 6. Modelo de calidad FURPS

Factor de Calidad	Atributos
Funcionalidad	• Características y capacidades del programa • Generalidad de las funciones • Seguridad del sistema

Factor de Calidad	Atributos
Facilidad de uso	• Factores humanos • Factores estéticos • Consistencia de la interfaz • Documentación
Confiabilidad	• Frecuencia y severidad de las fallas • Exactitud de las salidas • Tiempo medio de fallos • Capacidad de recuperación ante fallas • Capacidad de predicción
Rendimiento	• Velocidad del procesamiento • Tiempo de respuesta • Consumo de recursos • Rendimiento efectivo total • Eficacia
Capacidad de Soporte	• Extensibilidad • Adaptabilidad • Capacidad de pruebas • Capacidad de configuración • Compatibilidad • Requisitos de instalación

Fuente: (Dirección General de Servicio Civil, 2013)

2.2.13.4. Modelo ISO 9126

El estándar ISO/IEC 9126 fue desarrollado en un intento de identificar los atributos clave de calidad para un producto de software (Pressman, 2010). Este estándar es una simplificación del modelo McCall e identifica seis características básicas de calidad que pueden ser presentadas en cualquier producto software. En la figura n.1 se pueden apreciar estas seis características y sus sub características.

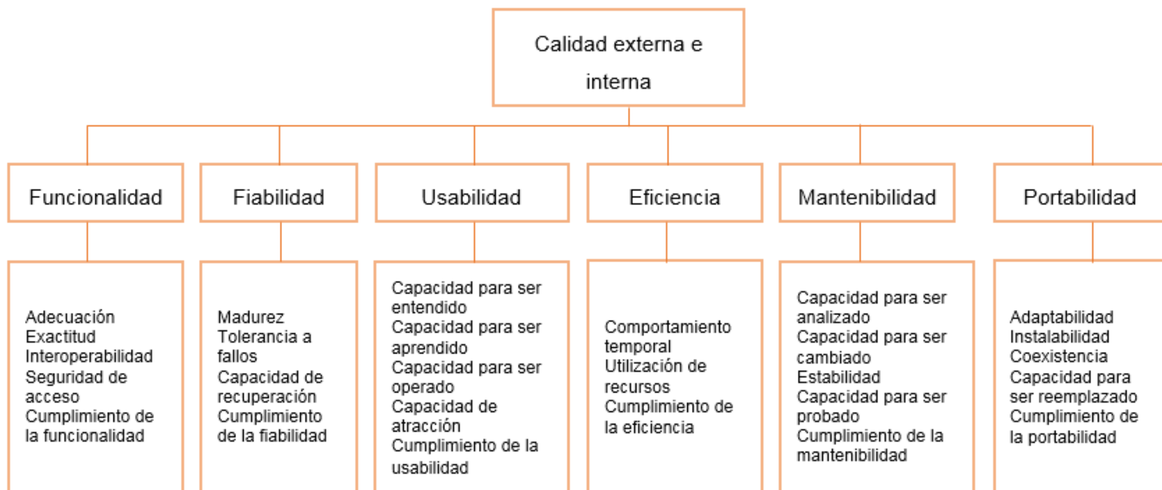


Figura n.° 1. Modelo ISO 2196

Fuente: (International Organization for Standardization, 2015)

2.2.13.5. Modelo Boehm

El modelo de Boehm agrega algunas características a las existentes en el modelo de McCall y representa una estructura jerárquica de características, cada una de las cuales contribuye a la calidad total. Consiste en un modelo de descomposición de características de calidad del software en 3 niveles previos a la aplicación de métricas.

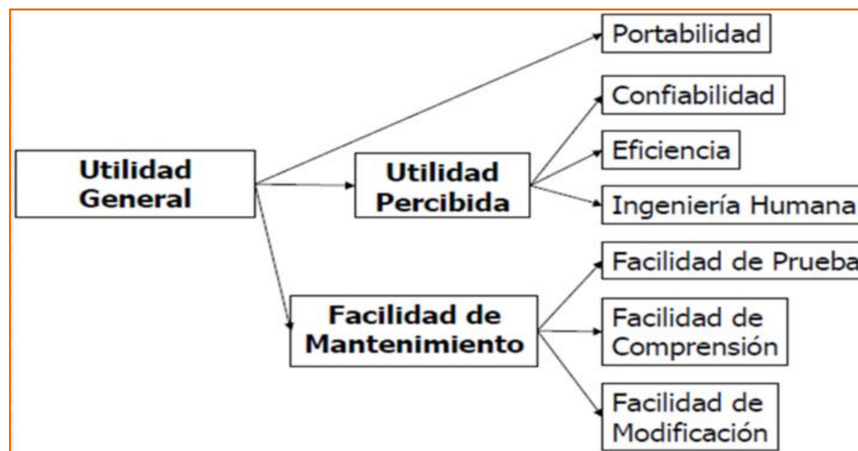


Figura n.° 2. Modelo de calidad de Boehm

Fuente: (Dirección General de Servicio Civil, 2013)

2.2.14. Métricas del producto

Everall Mills (1998) nos presenta un grupo de métricas utilizadas para medir la calidad de un software.

- Métricas de tamaño: un número de métricas intentan cuantificar el tamaño del software. La métrica más usada es, líneas de código (LOC). Una de sus desventajas es que sufre de valor hasta que el proceso de software haya finalizado.
- Métricas de complejidad: numerosas métricas son propuestas para medir la complejidad de un programa. A diferencia las métricas de tamaño, la métrica de complejidad puede ser calculada desde el inicio del ciclo del desarrollo del software.
- Métricas de calidad: estas métricas se centran en mejorar características de calidad del software para su corrección, eficiencia, portabilidad, mantenibilidad, confiabilidad, etc.

2.2.15. Cómo lograr la calidad del software

La calidad del software no solo se ve. Es el resultado de la buena administración del proyecto y una correcta práctica de la ingeniería de software (Pressman, 2010). La administración y la práctica se aplican en el contexto de cuatro actividades principales que ayuden al equipo de software a lograr una alta calidad de éste: métodos de ingeniería de software, técnicas de administración de proyectos, acciones de control de calidad y aseguramiento de la calidad del software.

- Métodos de la ingeniería de software: si se quiere construir un software de alta calidad, se debe ser capaz de crear un diseño que esté de acuerdo con el problema y que al mismo tiempo tenga características que llevan al software a las dimensiones y factores de calidad.
- Técnicas de administración de proyectos: si se tiene una buena planeación de riegos, entonces la calidad del software se verá influenciada de manera positiva.
- Control de calidad: el control de calidad incluye un conjunto de acciones de ingeniería de software que ayuden a asegurar que todo producto del trabajo cumpla sus metas de calidad. El código se inspecciona con objeto de descubrir y corregir errores antes de que comiencen las pruebas.
- Aseguramiento de la calidad: el aseguramiento de la calidad establece la infraestructura de apoyo a los métodos sólidos de la ingeniería de software, además

consiste de un conjunto de funciones de auditoría y reportes para evaluar la eficacia y completitud de las acciones de control de calidad.

2.2.16. Factor de calidad elegido

Haciendo un resumen de todo lo mencionado en apartados anteriores, notamos que la calidad de un producto software se determina en base a seis características, las cuales son: funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad. Asimismo tenemos los modelos de calidad del software, los cuales agrupan factores o métricas de calidad, algunos de estos factores parecidos y otros distan de las seis características de la calidad mencionadas, pero en conclusión todos estos factores se centran en lo mismo.

Ahora, si tratamos de abarcar todos estos factores o características de un producto software de calidad, harán que la investigación se vuelva muy extensa, lo cual impactaría considerablemente en el alcance del proyecto; dicho esto, solo haremos énfasis en el factor de mantenibilidad; esto por el impacto que este factor aporta en la calidad de un producto software, tal cual lo menciona (Juran, 1992). La calidad del software es afecta por dos grandes grupos:

- Los que miden directamente (defectos descubiertos en las pruebas).
- Los que se miden directamente (facilidad de uso o de mantenimiento).

En cada caso debe presentarse una medición, se debe comparar el software con algún conjunto de datos y obtener algún indicio sobe la calidad. Es por esto que se eligió herramientas que miden métricas asociadas a la mantenibilidad de un producto software.

2.2.17. Dimensión de las variables

Para el dimensionamiento de las variables tenemos:

- Herramienta software de calidad: tipo de herramienta, métricas que evalúan y facilidad de uso; que son aspectos básicos no solamente de este tipo de herramientas sino de manera global.
- Software de calidad: para lograr esto, se eligió las siguientes métricas de calidad:
  - Complejidad ciclomática, aquí tenemos a (Pressman, 2010), quien hace énfasis en esta métrica para lograr tener calidad en el código fuente de un software.
  - Líneas de código, (Fowler, 1999) claramente nos habla sobre esta métrica para lograr la construcción de un producto software de calidad.

- Cobertura de código, del mismo modo que la métrica anterior, (Fowler, 1999) pone énfasis en esta métrica como importante para lograr construir software de calidad.
- Código repetido, según Martin Fowler (1999), esta métrica es el peor enemigo de la mantenibilidad ya que aumenta innecesariamente el número de líneas de código, dispara los costes y aumenta los riesgos en la funcionalidad del software.
- Deuda técnica, Fontana et al.(2016) nos mencionan que esta métrica a la larga influye no solo en la calidad del software, sino también en el tiempo que implicaría corregir los problemas presentados por culpa de un desarrollo apresurado.

2.3. Hipótesis

El uso de herramientas de calidad produce un impacto considerablemente positivo en la implementación de software de calidad.

CAPÍTULO 3. METODOLOGÍA

3.1. Operacionalización de variables

Tabla n.° 7. Operacionalización de variables

Variable	Definición Conceptual	Dimensiones	Indicadores
Herramienta software de Calidad	Herramientas que realizan un control desde el punto de vista estático, es decir, analizan las fuentes del software sin la necesidad de ejecutarlo (Garzas, 2012).	Tipo de herramienta	% de indicadores abarcados
		Métricas a evaluar	Número de métricas evaluadas
		Facilidad de uso	Nivel de facilidad de uso
			Nivel de aprendizaje
Software de Calidad	El software que cumple con los requerimientos del negocio, proporciona al usuario una experiencia gratificante, y tiene menos defectos (Wailgum, 2010).	Código repetido	% de calidad
			% de código duplicado
			% de líneas duplicadas
		Complejidad ciclomática	% de condiciones permitidas
			% de complejidad por método
			% de complejidad total
			Nivel de mantenibilidad
			Nivel de complejidad
		Cobertura de código	% de cobertura de pruebas
		Líneas de código	% máximo de líneas de código por método

Fuente: Elaboración propia

3.2. Metodologías

3.2.1. Metodología de desarrollo de software

3.2.1.1. ¿Por qué una metodología ágil?

El término “ágil” generalmente se refiere a ser capaz de moverse o responder de forma rápida y fácil; a ser ágil. Por lo tanto, en cualquier tipo de disciplina de la gerencia, ser ágil es una calidad que se debe tratar de alcanzar. Específicamente, la ejecución de proyectos ágiles implica ser adaptable durante la creación de un producto, servicio y otro resultado. Es importante comprender que mientras que los métodos ágiles de desarrollo son altamente adaptables, también es necesario considerar la estabilidad en sus procesos de adaptación (SCRUMstudy, 2016).

Una metodología ágil es una de las mejores formas de desarrollar software, ya que se enfatiza en lo siguiente (Cunningham, 2001):

- Los individuos e interacciones por encima de los procesos y herramientas.
- Un software funcional por encima de la documentación extensiva.
- Colaboración con el cliente por encima de la negociación contractual.
- El responder al cambio por encima de seguir un plan.

Con una metodología ágil:

- Se aumenta el retorno sobre la inversión.
- Se entrega resultados fiables.
- Se da rienda suelta a la creatividad y la innovación.
- Se aumenta el rendimiento a través de la orientación del grupo.
- Se mejora la eficacia y fiabilidad a través de estrategias, procesos y prácticas.

3.2.1.2. Metodologías de desarrollo de software

Scrum: Scrum es una de las maneras más populares para poner en práctica ágil. Se trata de un modelo de software iterativo que sigue un conjunto de roles, responsabilidades y reuniones que nunca cambian. Sprint, que suele durar una o dos semanas, permite que un equipo de entregar el software de forma regular; la figura n.° 3 muestra el flujo del proceso seguido por Scrum.

Figura n.° 3. Metodología Scrum

Fuente: (Ghahrai, 2016)

Kanban: significa "signo visual" o "tarjeta" en japonés, es un marco visual para poner en práctica ágil. Promueve cambios pequeños y continuos a su sistema actual. Sus principios son: la visualización del flujo de trabajo, limitar el trabajo en curso, gestionar y mejorar el flujo, hacer las políticas explícitas, y mejorar continuamente; a continuación se observa la figura n.° 4 con un tablero Kanban básico.

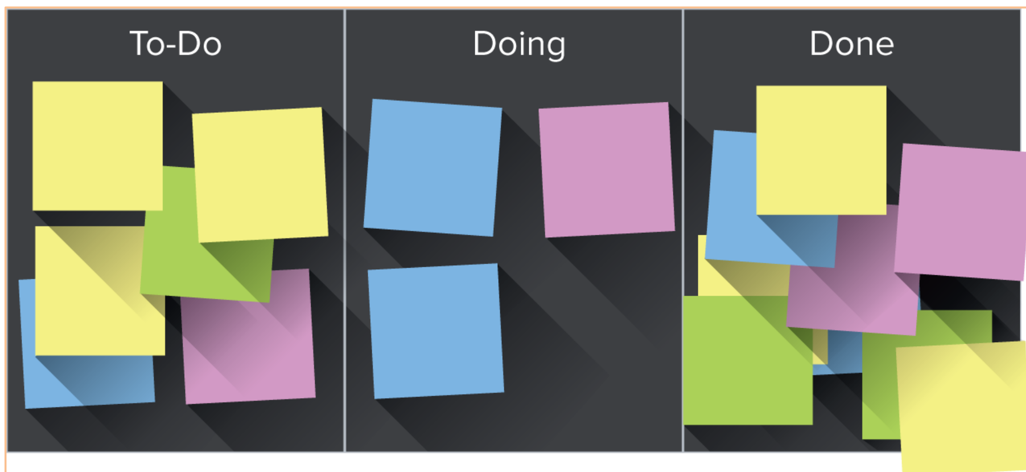


Figura n.° 4. Tablero de Kanban

Fuente: (Smartsheet, 2016)

XP: También conocido como XP, la programación extrema es un tipo de desarrollo de software destinado a mejorar la calidad y capacidad de respuesta a las cambiantes necesidades de los clientes. Los principios de XP incluyen comentarios, asumiendo la sencillez y abrazar el cambio. A continuación se observa la figura n.° 5 con el flujo de proceso de XP.

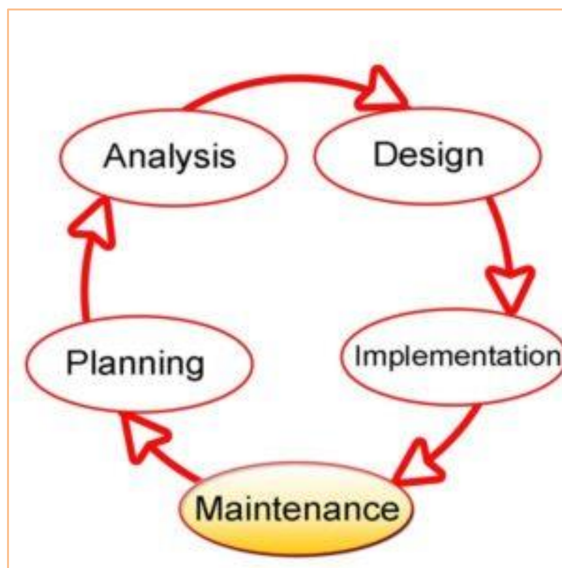


Figura n.° 5. Metodología XP

Fuente: (Ghahrai, 2016)

3.2.1.2.1 Tabla comparativa de metodologías de desarrollo de software

Tabla n.° 8. Tabla comparativa de metodologías de desarrollo de software

	SCRUM	**KANBAN**	**XP**
Objetivo	Uso de equipos multifuncionales auto organizados y empoderados que dividen su trabajo en ciclos de trabajo cortos y concentrados denominados Sprints.	Reduce los impedimentos que llevan a tomar más tiempo en la entrega del producto, no las piezas necesarias del proceso.	Organiza a la gente para producir software de mayor calidad y de manera más productiva.
Especialidad	Una fuerza clave de Scrum reside en el uso de equipos multifuncionales, auto organizado y empoderado que dividen su trabajo en ciclos de trabajo cortos y	En Kanban se visualiza el flujo de trabajo: el trabajo se divide en artículos pequeños y discretos y se escribe en una tarjeta pegada a un tablero. El tablero tiene diversas	El éxito radica en el hincapié que se hace a la satisfacción del cliente.

	SCRUM	KANBAN	XP
	concentrados denominados Sprints.	columnas y cómo el trabajo progresa a través de diversas etapas.	
Valores	• Foco • Coraje • Opennes • Compromiso • Respeto	• Transparencia • Acuerdo • Equilibrio • Respeto • Comprensión • Liderazgo • Colaboración • Atención al cliente • Flujo	• Comunicación • Simplicidad • Retroalimentación • Coraje • Respeto
Roles	Funciones básicas: Product Owner, Scrum Master, Scrum Team. Funciones no básicas: Stakeholder, Scrum Guidance, Vendors, Jefe Scrum Master	No hay roles existentes.	Tracker, cliente, programador, entrenador, gerente, tester
Prácticas	• Planificación • Scrum diario • Revisión y retrospectiva • Extensión Backlog • Artefactos	• Visualizar • Limitar el trabajo en progreso • Administrar el flujo • Hacer implícitas las políticas de gestión • Mejorar de manera colaborativa	• Retroalimentación • Proceso continuo • Conocimiento compartido • Bienestar del programador

Fuente: (Kulikova, 2016)

3.2.1.3. Elección de la metodología.

Como metodología elegida para la implementación de las mejoras en el código fuente del sistema SISGGED, tenemos a Kanban, debido a que no hay roles ni estructura organizativa, se evita los cuellos de botella, en especial considerando que hay un solo Tesista, se evita los cuellos de botellas en la resolución de las tareas, no se necesita priorizar las tareas, se cuenta con un tablero

para visualizar el flujo de trabajo y limitar las obras en curso y, porque Kanban es adecuado para equipos con recursos limitados (Bowes, 2015).

3.2.1.4. Fases de Kanban

En resumen, se considera que son 4 las fases principales para una buena implantación del sistema Kanban, y estas son (Angeles Estrada, 2006):

- Fase 1. Entrenar a todo el personal en los principios de Kanban, y los beneficios de usar Kanban.
- Fase 2. Implementar Kanban en aquellos componentes con más problemas para facilitar su manufactura y para resaltar los problemas escondidos. El entrenamiento con el personal continúa en la línea de producción.
- Fase 3. Implementar Kanban en el resto de los componentes, esto no debe ser problema ya que para esto, los operadores ya han visto las ventajas de Kanban.
- Fase 4. Esta fase consiste en la revisión del sistema Kanban, los puntos de reorden y los niveles de reorden.

3.2.1.5. Actividades de Kanban

Las actividades básicas realizadas en un flujo de trabajo básico de Kanban son: (Kulikova, 2016):

- Por hacer
- En Progreso
- Hecho

3.2.1.6. ¿Por qué no las demás metodologías?

- Scrum: se llena el tablero con todas las tareas necesarias para acabar un ciclo de sprint. Al final del sprint, todas las tareas deberán están en estado Done. Los Sprint ayudan al cliente a definir qué es lo más importante y lo que realmente quiere que se desarrolle. No se requiere tantas reuniones ni roles.
- Lean: busca eliminar todo lo que no aporte valor, en nuestra investigación todas las mejoras hechas al código aportan valor.
- XP: necesita algunos roles del mismo modo que Scrum.

3.2.2. Desarrollo de la metodología del producto de aplicación profesional

Actualmente, los equipos de desarrollo de software ágil pueden aprovechar esos mismos principios de justo a tiempo (JIT) ajustando la cantidad de trabajo en curso (WIP) a la capacidad

del equipo. Los equipos tienen así opciones de planificación más flexibles, resultados más rápidos, un enfoque más claro y transparencia a lo largo del ciclo de desarrollo (Atlassian, 2017).

El ciclo de desarrollo será dividido en 7 ciclos de medición de los indicadores con la ayuda de las herramientas de calidad a utilizar.

Se utilizará como herramienta de apoyo a Atlassian JIRA para la visualización del tablero Kanban, gracias a su facilidad de uso.

Las herramientas de calidad a utilizar son:

- SonarQube
- Designite
- Visual Studio Code Metrics
- Cloc

Los ciclos en los cuales se realizará las mediciones son:

- Pre test
- Ciclo 1
- Ciclo 2
- Ciclo 3
- Ciclo 4
- Ciclo 5
- Post test

Para realizar el cálculo del total de ciclos de medición se tomó como base algunas características utilizadas en la metodóloga de desarrollo Kanban, estas son:

- Estimación total de tarjetas Kanban: 152 horas aproximadamente.
- Esfuerzo dedicado por semana: 20 horas por miembro del equipo.
- Equivalencia de un sprint (ciclo): 1 semana por ser sprint corto.

Por lo tanto, aplicando una división tenemos:

$$\frac{152 \text{ horas}}{20 \text{ horas}} = 7.6 \text{ semanas}$$

Como una semana equivale a un sprint, se tiene que el total de sprint o ciclos de medición será 7.

3.2.2.1. Herramienta de apoyo

3.2.2.1.1 JIRA Software

JIRA Software es una herramienta ágil de gestión de proyectos compatible con cualquier metodología ágil, ya sea Scrum, Kanban o la tuya propia. Desde tableros hasta informes ágiles, puedes planificar, supervisar y gestionar todos los proyectos de desarrollo de software ágil con una sola herramienta. Elige una metodología para ver cómo JIRA Software puede hacer que tu equipo publique software de calidad con mayor rapidez (Atlassian, 2017).

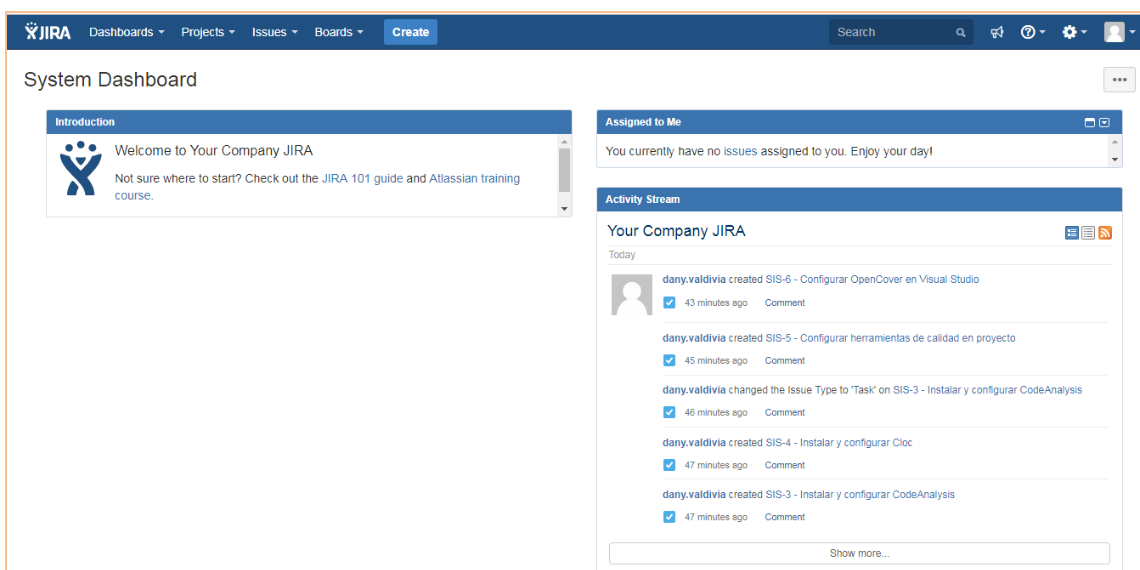


Figura n.° 6. Dashboard de JIRA Software

Fuente: (Atlassian, 2017)

3.2.2.1.1.1 Tablero Kanban

Una de la características principales de la elección de JIRA Software es su tablero de Kanban, el cual es ideal en nuestro caso, ya que el equipo está conformado por un solo miembro; debido a esto se tiene que restringir el trabajo en progreso. A continuación podemos ver un tablero Kanban utilizado para nuestro proyecto. (Atlassian, 2017)

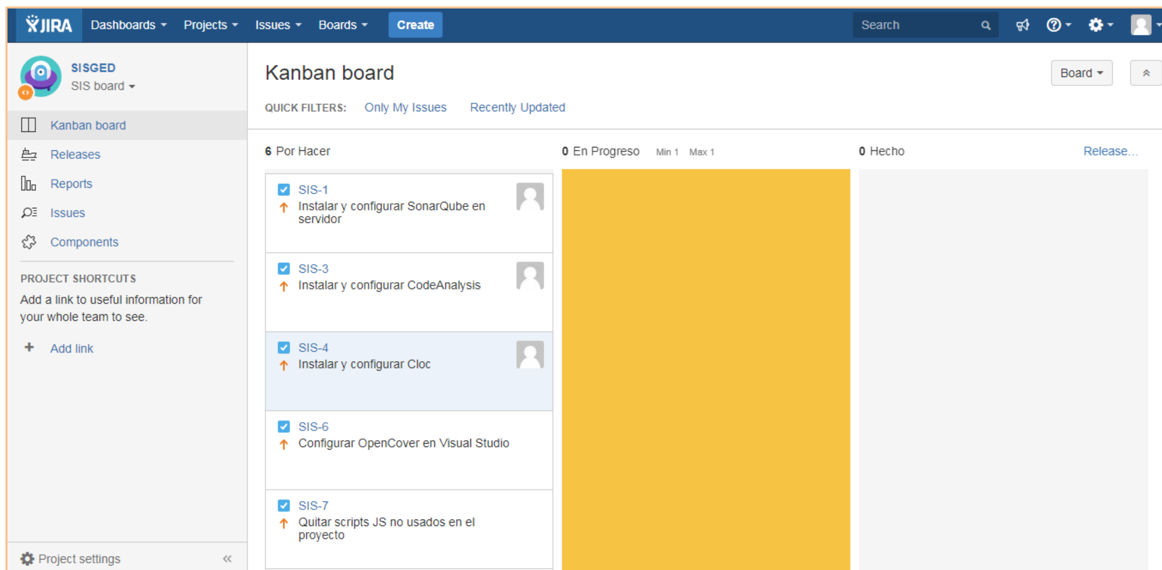


Figura n.° 7. Tablero Kanban para el proyecto SISGED

Fuente: Elaboración propia

3.2.2.1.1.2 WIP en JIRA

Asimismo podemos configurar nuestro tablero Kanban para limitar el trabajo en progreso, de modo que nos muestre una advertencia cuando se tenga asignada más de una tarea a un miembro del equipo.

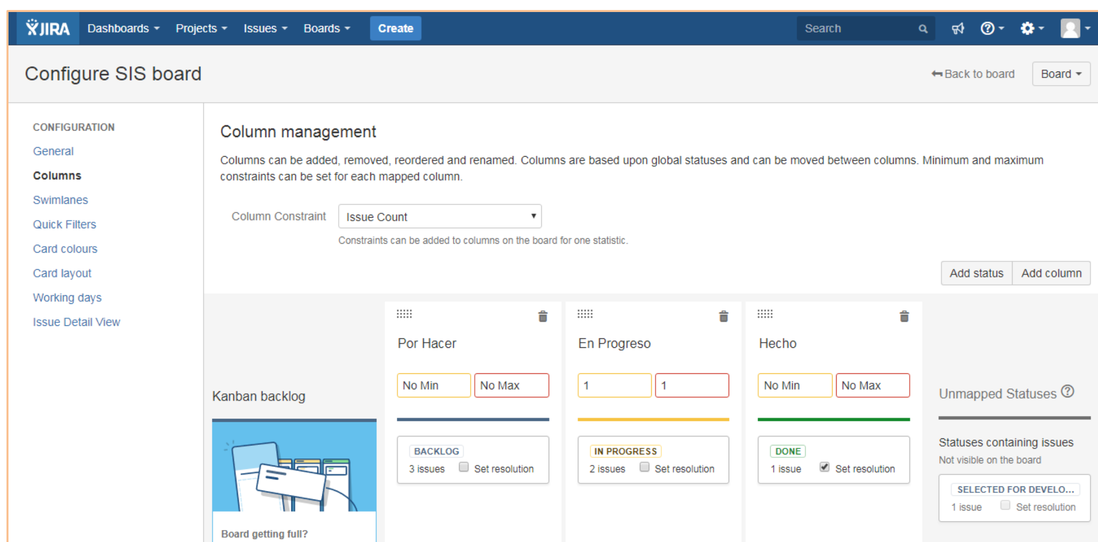


Figura n.° 8. Pantalla de configuración de WIP en tablero Kanban

Fuente: Elaboración propia

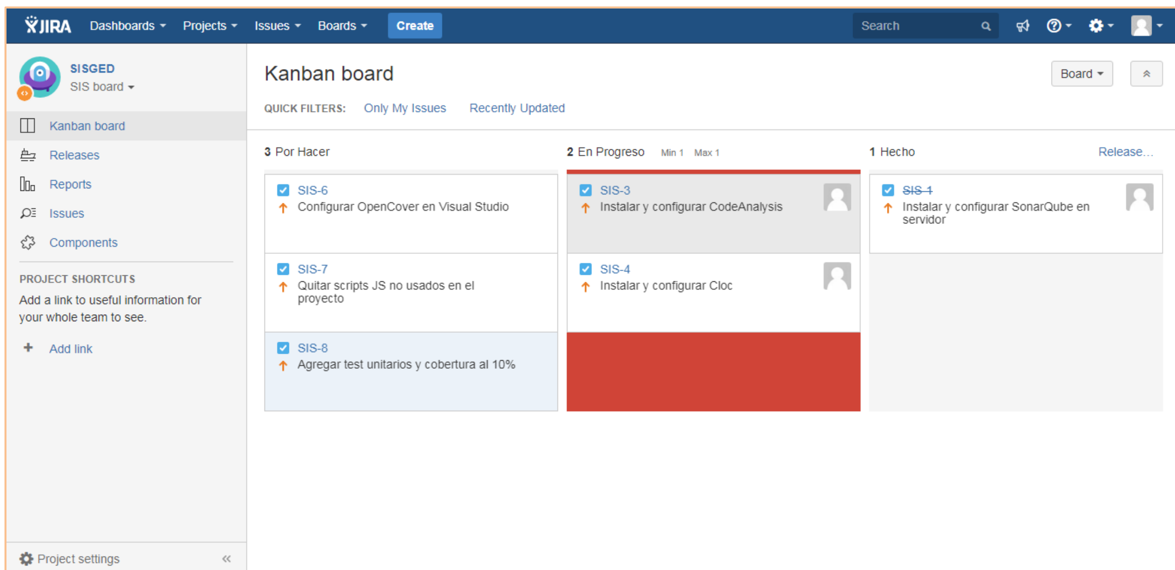


Figura n.° 9. Visualización de tarjetas Kanban en JIRA

Fuente: Elaboración propia

3.2.2.1.1.3 Tarjetas Kanban

Para los equipos de Kanban, cada elemento de trabajo se representa en una tarjeta distinta del tablero.

El objetivo principal de representar el trabajo como una tarjeta en el tablero de Kanban es que los miembros del equipo realicen el seguimiento del progreso del trabajo mediante el workflow de una manera muy visual. Las tarjetas Kanban presentan información vital sobre ese elemento de trabajo concreto y proporcionan una visibilidad completa a todo el equipo sobre quién está a cargo de ese elemento de trabajo concreto, una breve descripción del trabajo que se está haciendo, la duración estimada que llevará esa unidad de trabajo, etc (Atlassian, 2017).

Figura n.° 10. Tarjeta Kanban en JIRA

Fuente: Elaboración propia

3.2.2.2. Herramientas de calidad

Aquí tenemos las herramientas software utilizadas en la presente investigación, las cuales nos facilitarán y asegurarán mejorar la calidad del software del sistema SISGED:

3.2.2.2.1 SonarQube

SonarQube es una plataforma de código abierto usada por los equipos de desarrollo para controlar la calidad del código. SonarQube fue desarrollado con el principal objetivo de hacer accesible la administración de la calidad del código con un mínimo esfuerzo. Como tal, Sonar contiene en su núcleo de funcionalidades un analizador de código, una herramienta de reportes, un módulo que detecta defectos y una función para regresar los cambios realizados en el código. Prácticamente en todas las industrias, los grandes líderes utilizan métricas. Ya se trate de defectos y pérdidas en la manufactura, ventas y ganancias, o la tracción en una página web, existen esas métricas que

les dice a los líderes como están haciendo su trabajo y que tan efectivo es, en general si se están obteniendo mejores o peores resultados. Ahora existen este tipo de métricas para el desarrollo de software, empaquetadas y presentadas a través de una plataforma estandarizada, basada en servidor (sin necesidad que el cliente instale algo) que utiliza herramientas respetadas en la industria, como Findbug, PMD y JaCoCo. Esos resultados son presentados de forma intuitiva por medio de una interfaz web que además ofrece RSS de las alertas generadas cuando los umbrales de calidad establecidos se vulneran. En términos de lenguajes SonarQube soporta Java en su núcleo principal, pero es flexible a través de plugins comerciales o libres, por lo tanto se puede extender a lenguajes como Actionscript, PHP, PL/SQL, Python, etc. ya que el motor de reportes es independiente del lenguaje de análisis. A continuación se muestra una lista con los lenguajes que soporta SonarQube (Ospina, 2015).

Tabla n.° 9. Lenguajes de programación soportados por SonarQube

Lenguajes	Pago/Gratis	Métricas
Abap	Pago	Tamaño, comentarios, complejidad, duplicados y errores
C	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
C++	Gratis/ Pago	Tamaño, comentarios, complejidad, duplicados y errores (También existen plugins de pago).
C#	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas, errores. El análisis de C# se ejecuta por una seria de herramientas externas que deben ser instaladas de forma individual.
Cobol	Pago	Tamaño, comentarios, complejidad, errores. Métricas especificas del lenguaje como salida y entrada de declaraciones.
Delphi	Gratis	Tamaño, comentarios, complejidad, diseños, pruebas duplicados y errores
Drools	Gratis	Tamaño, comentarios, y errores
Flex/Actionscript	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas y errores
Groovy	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas y errores
Javascript	Gratis	Tamaño, comentarios, complejidad, duplicados, pruebas y errores
Natural	Pago	Tamaño, comentarios, complejidad, duplicados y errores
Php	Gratis	Tamaño, comentarios, complejidad, duplicados y errores

Lenguajes	Pago/Gratis	Métricas
PL/1	Pago	Tamaño, comentarios, complejidad, duplicados y errores
PL/SQL	Pago	Tamaño, comentarios, complejidad, duplicados y errores
Python	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
Visual Basic 6	Pago	Tamaño, comentarios, complejidad, duplicados y errores
JSP	Gratis	Tamaño, comentarios, complejidad, duplicados y errores
XML	Gratis	Tamaño y errores

Fuente: (Ospina, 2015)

3.2.2.2.1.1 Instalación y configuración de SonarQube

Para ejecutar el primer análisis de nuestro código fuente, es requisito tener configurado SonarQube en donde se almacenarán los resultados de nuestros análisis y asimismo configurar SonarQube Scanner for MSBuild, el cual se encargará de enviar el resultado de los análisis hacia SonarQube. Para la instalación de SonarQube y SonarQube Scanner for MSBuild se deben seguir los pasos especificados en los apéndices A y B respectivamente.

3.2.2.2.1.2 Configuración de proyecto en SonarQube

Lo siguiente es realizar la configuración del proyecto en SonarQube, para lo cual se accede a la sección de administración de proyectos. Una vez dentro, seguir los siguientes pasos:

1. Hacer clic en Create Project.
2. Completar los campos tal como se visualiza en la figura n.° 11.

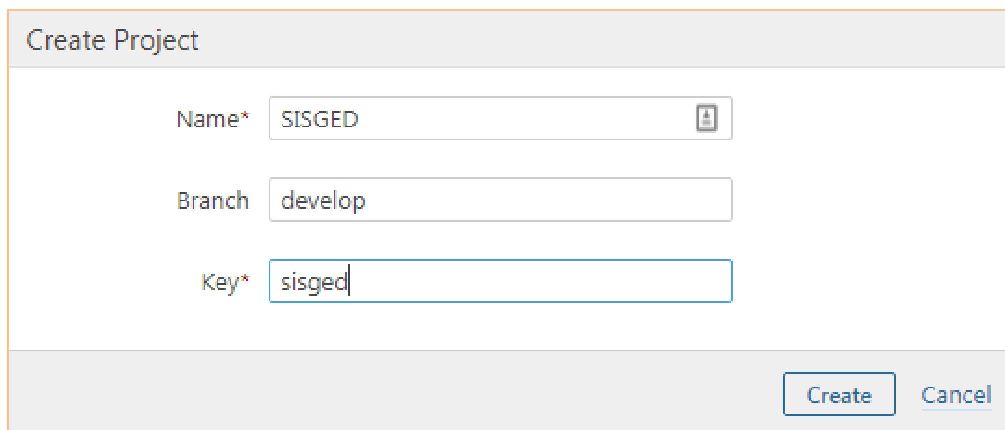


Figura n.° 11. Configuración de proyecto en SonarQube

Fuente: Elaboración propia

3. Hacer clic en Create.

Luego de creado el proyecto, se puede visualizar la lista de proyectos existentes en SonarQube.

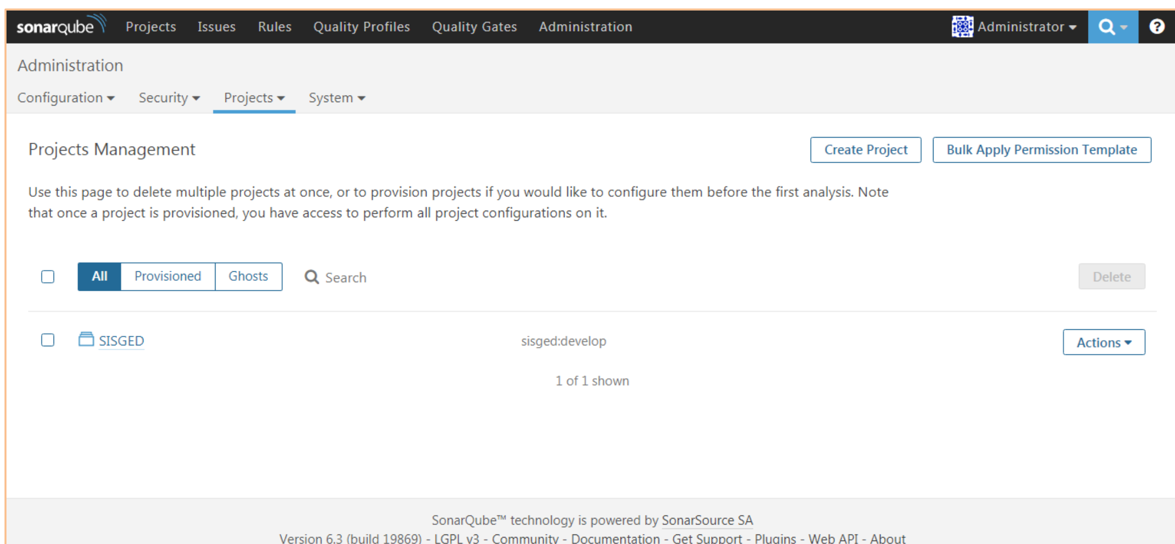


Figura n.° 12. Administración de proyectos en SonarQube

Fuente: Elaboración propia

3.2.2.2.1.3 Ejecución del análisis

Para ejecutar un análisis con SonarQube, se debe agregar un archivo llamado sonar-project.properties a la ruta principal del proyecto; el contenido del archivo debe ser similar al siguiente:

```
sonar.projectKey=sisged:develop
sonar.projectName=SISGED
sonar.projectVersion=1.0
sonar.sources=.
sonar.exclusions=UPNSAC.Sisged.Test/obj/**, non-nugget-packages/**, packages/**, UPNSAC.Sisged.BE/obj/**, UPNSAC.Sisged.BL/obj/**, UPNSAC.Sisged.DAO/obj/**, UPNSAC.Sisged.WebApp/obj/**, UPNSAC.Sisged.WebApp/Scripts/*.min.js, UPNSAC.Sisged.WebApp/Content/*.min.css, UPNSAC.Sisged.DataBase/*

sonar.sourceEncoding=UTF-8
```

Una vez agregado el archivo, abrir la consola de Windows en la ruta principal del proyecto y ejecutar los comandos

```
MSBuild.SonarQube.Runner.exe begin /k:"sisged:develop" /n:"SISGED" /v:"1.0" /d:sonar.exclusions="file:D:\Projects\sisged\UPNSAC.Sisged.WebApp\Scripts\Plugins/**/*.*" /d:sonar.cs.opencover.reportsPaths="%CD%\opencover.xml"

msbuild

OpenCover.Console.exe -output:"%CD%\opencover.xml" -target:nunit-console.exe -targetargs:"/nologo /noshadow D:\Projects\sisged\UPNSAC.Sisged.Test\bin\Debug\UPNSAC.Sisged.Test.dll" -filter:"+[*]* -[Ihavethetests*]*" -register:user

MSBuild.SonarQube.Runner.exe end
```

Luego de ejecutado el último comando, el análisis iniciará de forma inmediata, al final en la consola se mostrará el tiempo que tomó el análisis en procesarse, este tiempo es variable dependiendo al tamaño del proyecto. A continuación se muestra la salida en consola de un análisis realizado con SonarQube Scanner for MSBuild.

```
INFO: ------------- Scan SISGED
INFO: Initializer GenericCoverageSensor
INFO: Initializer GenericCoverageSensor (done) | time=0ms
INFO: Base dir: D:\Projects\sisged
INFO: Working dir: D:\Projects\sisged\.sonarqube\out\.sonar
INFO: Source encoding: windows-1252, default locale: es_PE
INFO: Sensor NoSonar Sensor [php]
INFO: Sensor NoSonar Sensor [php] (done) | time=2ms
INFO: Sensor Coverage Report Import [csharp]
INFO: Parsing the OpenCover report D:\Projects\sisged\opencover.xml
INFO: Adding this code coverage report to the cache for later reuse: D:\Projects\sisged\opencover.xml
INFO: Sensor Coverage Report Import [csharp] (done) | time=102ms
INFO: Sensor Coverage Report Import [csharp]
INFO: Sensor Coverage Report Import [csharp] (done) | time=0ms
INFO: Sensor Unit Test Results Import [csharp]
INFO: Sensor Unit Test Results Import [csharp] (done) | time=1ms
INFO: Sensor XmlFileSensor [java]
INFO: Sensor XmlFileSensor [java] (done) | time=0ms
INFO: Sensor Analyzer for "php.ini" files [php]
INFO: Sensor Analyzer for "php.ini" files [php] (done) | time=0ms
INFO: Sensor Zero Coverage Sensor
INFO: Sensor Zero Coverage Sensor (done) | time=0ms
INFO: Sensor Code Colorizer Sensor
INFO: Sensor Code Colorizer Sensor (done) | time=0ms
INFO: Sensor CPD Block Indexer
INFO: Sensor CPD Block Indexer (done) | time=1ms
INFO: Calculating CPD for 308 files
INFO: CPD calculation finished
INFO: Analysis report generated in 11515ms, dir size=2 MB
INFO: Analysis reports compressed in 2300ms, zip size=1 MB
INFO: Analysis report uploaded in 12859ms
INFO: ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/sonar/dashboard/index/sisged:develop
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
INFO: More about the report processing at http://localhost:9000/sonar/api/ce/task?id=AV4176sdvVqUMoCoxRj7
INFO: Task total time: 1:06.727 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 1:15.812s
INFO: Final Memory: 53M/429M
INFO: ------------------------------------------------------------------------
The SonarQube Scanner has finished
20:37:50.149  Creating a summary markdown file...
20:37:50.152  Analysis results: http://localhost:9000/sonar/dashboard/index/sisged:develop
20:37:50.153  Post-processing succeeded.
```

Figura n.° 13. Salida consola análisis con SonarQube Scanner for MSBuild

Fuente: Elaboración propia

3.2.2.2.1.4 Dashboard de SonarQube

Una vez completado el análisis, podemos observar las métricas de calidad en el Dashboard de SonarQube. Para lo cual debemos acceder a http://localhost:9000/sonar.

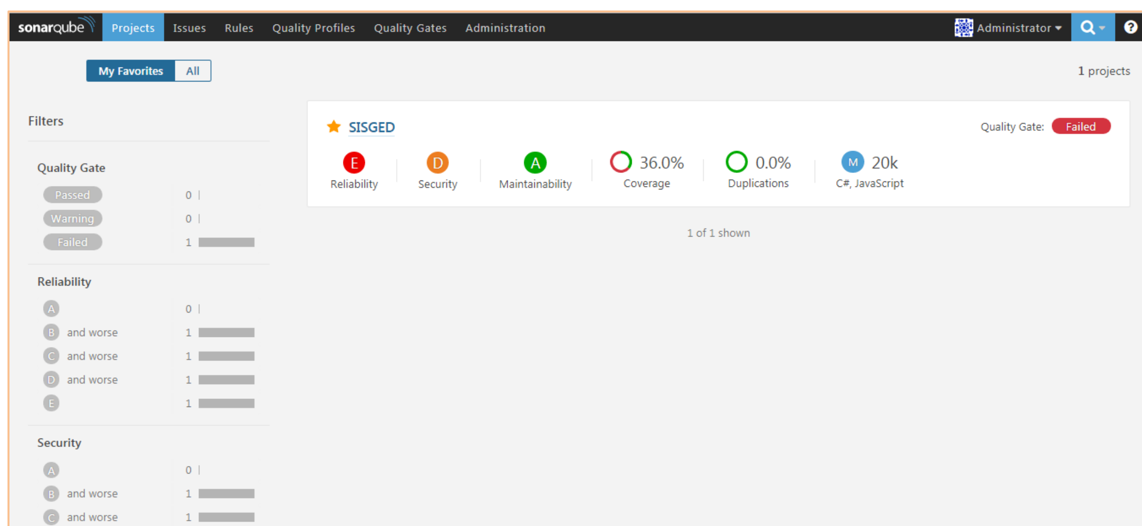


Figura n.° 14. Dashboard SonarQube

Fuente: Elaboración propia

En la Figura n.° 14 se puede observar la página principal de SonarQube, en la cual se visualiza filtros para ubicar proyectos según sus rating y además la lista de proyectos analizados. A nivel de cada proyecto analizado podemos visualizar el rating de acuerdo a cada categoría como son: confiabilidad, seguridad, mantenibilidad, cobertura y duplicados.

Para más detalle de las mediciones ingresamos al proyecto haciendo clic en el nombre de este y podemos observar las mediciones de los indicadores agrupados por categoría: bugs y vulnerabilidades, olores de código, cobertura y duplicados.

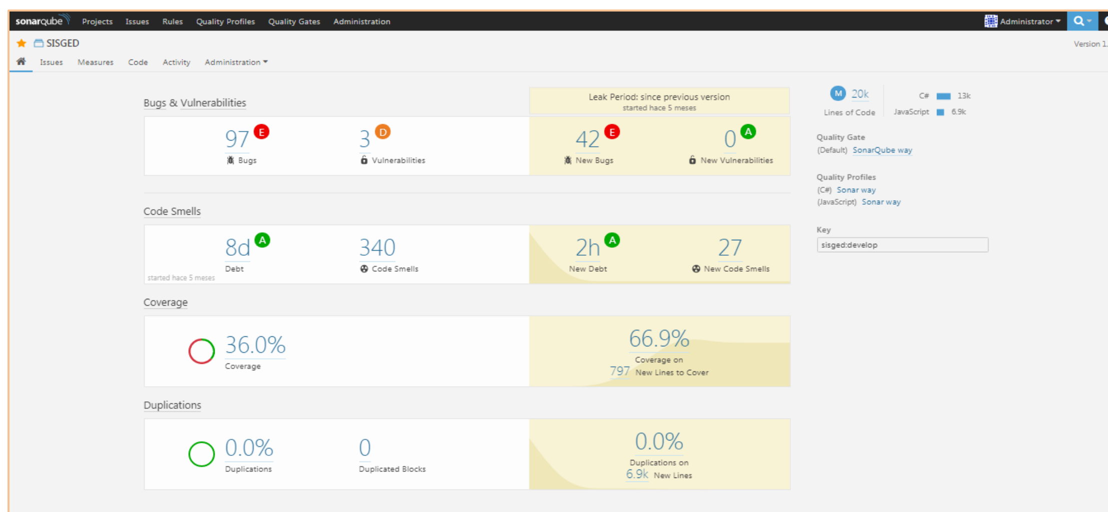


Figura n.° 15. Indicadores proyecto SISGED

Fuente: Elaboración propia

3.2.2.2.2 Designite

Designite es una herramienta diseñada específicamente para el análisis de código fuente escrito en C#. Esta herramienta toma el código fuente como entrada, lo analiza y presenta la información de evaluación de diseño interactivamente. Después del análisis, Designite presenta un resumen del análisis como se muestra en la Figura n.° 16.

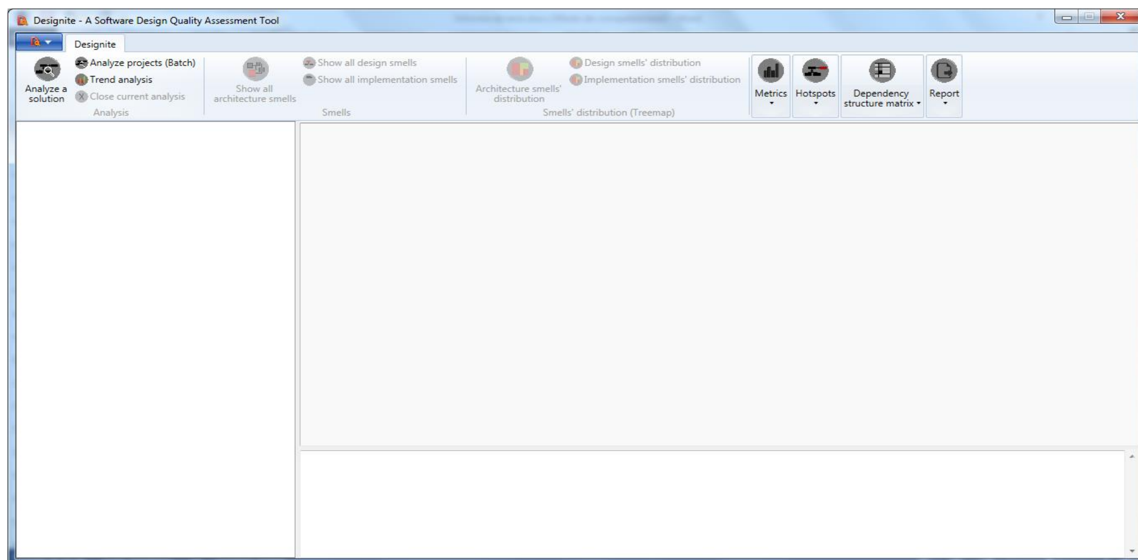


Figura n.° 16. Designite

Fuente: Elaboración propia

3.2.2.2.2.1 Instalación y configuración

Para ejecutar el análisis de nuestro código fuente, es requisito tener instalado Designite. Para la instalación de Designite se debe seguir los pasos especificados en el Apéndice C.

3.2.2.2.2.2 Ejecución de análisis

Los pasos para ejecutar el análisis son:

1. Abrir Designite.
2. Hacer clic en la opción **Analyze a Solution**.
3. Ubicar y abrir el archivo de solución a analizar.
4. Seleccionar los proyectos a analizar.
5. Hacer clic en **Analize**.
6. Una vez finalizado el análisis, el resultado es mostrado como la figura n.° 17.

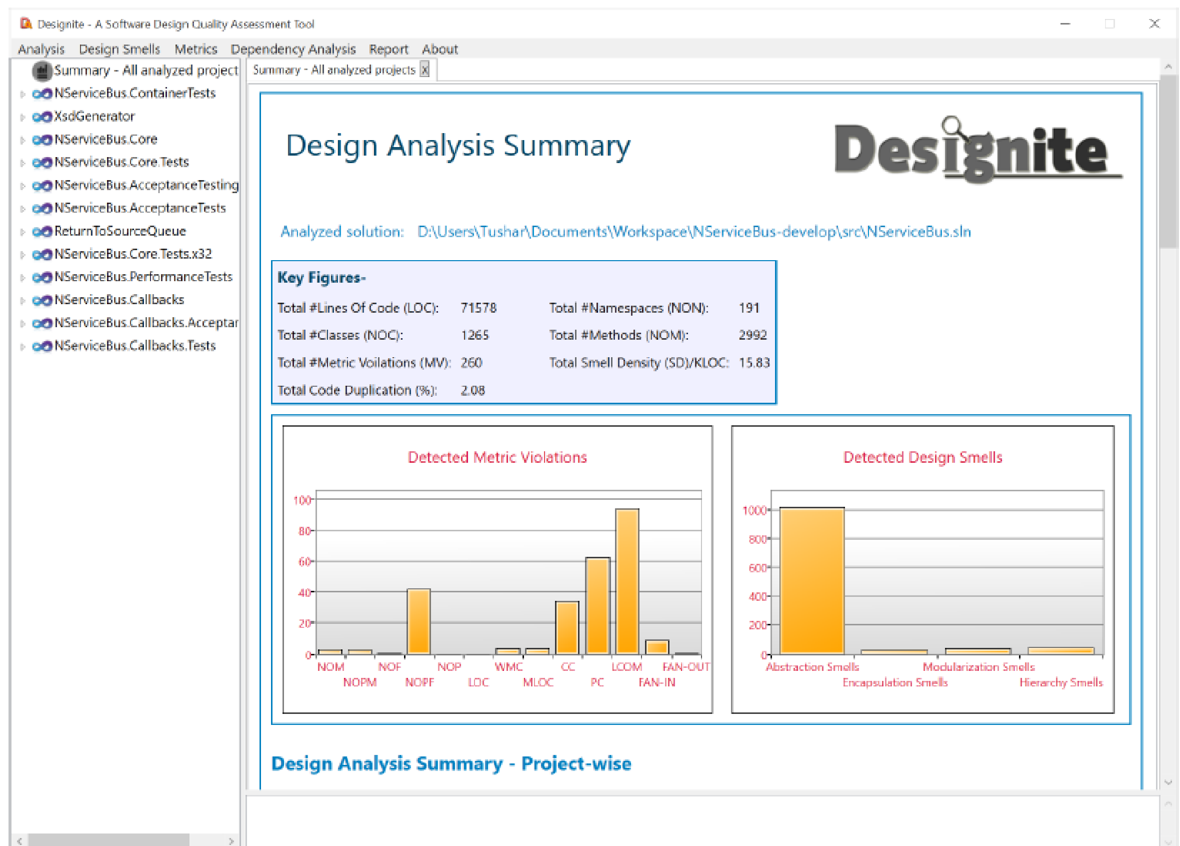


Figura n.° 17. Resumen de análisis Designite

Fuente: Elaboración propia

3.2.2.2.3 Visual Studio Code Metrics

Las métricas de código son un conjunto de medidas de software que proporcionan a los desarrolladores una mejor comprensión del código que están desarrollando. Aprovechando las métricas de código, los desarrolladores pueden entender qué tipos y/o métodos deben ser revisados o probados más a fondo. Los equipos de desarrollo pueden identificar riesgos potenciales, comprender el estado actual de un proyecto y seguir el progreso durante el desarrollo de software (Microsoft, 2017).

3.2.2.2.3.1 Instalación y configuración

Para poder contar con esta herramienta lo único que se debe hacer es tener instalado el IDE Visual Studio 2015.

3.2.2.2.3.2 Mediciones de software

La siguiente tabla muestra los resultados de métricas de código que Visual Studio Code Metrics calcula.

Tabla n.° 10. Resultados de métricas de código.

Métrica de código	Resultado
Índice de mantenibilidad	Calcula un valor de índice entre 0 y 100 que representa la facilidad de mantenimiento del código. Un valor alto significa una mejor mantenibilidad. Las clasificaciones codificadas con colores se pueden utilizar para identificar rápidamente puntos problemáticos en su código. Una calificación de verde está entre 20 y 100 e indica que el código tiene una buena capacidad de mantenimiento. Una calificación amarilla es entre 10 y 19 e indica que el código es moderadamente mantenible. Una clasificación roja es una clasificación entre 0 y 9 e indica una baja mantenibilidad.
Complejidad ciclomática	Mide la complejidad estructural del código. Se crea calculando el número de rutas de código diferentes en el flujo del programa. Un programa que tiene flujo de control complejo requerirá más pruebas para lograr una buena cobertura de código y será menos mantenible.
Profundidad de herencia	Indica el número de definiciones de clase que se extienden a la raíz de la jerarquía de clases. Cuanto más profunda sea la jerarquía, más difícil será comprender dónde se definen y / o redefinen métodos y campos particulares.
Acoplamiento de clases	Mide el acoplamiento a clases únicas a través de parámetros, variables locales, tipos de retorno, llamadas a métodos, instanciaciones genéricas o de plantilla, clases base, implementaciones de interfaces, campos definidos en tipos externos y decoración de atributos. Un buen diseño de software dicta que los tipos y métodos deben tener alta cohesión y bajo acoplamiento. Alto acoplamiento indica un diseño que es difícil de reutilizar y mantener debido a sus muchas interdependencias en otros tipos.
	Indica el número aproximado de líneas en el código. El recuento se

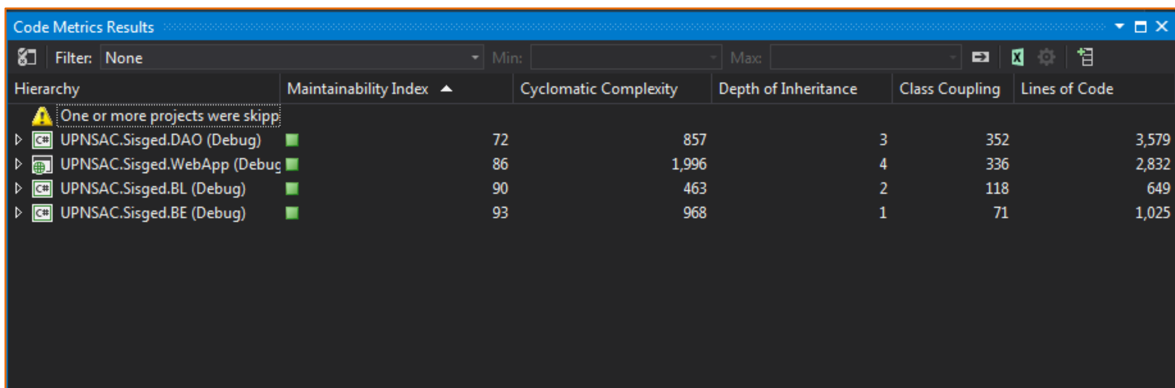
Líneas de código	basa en el código IL y, por lo tanto, no es el número exacto de líneas en el archivo de código fuente. Un recuento muy alto podría indicar que un tipo o método está tratando de hacer demasiado trabajo y debe dividirse. También podría indicar que el tipo o método podría ser difícil de mantener.

Fuente: Elaboración propia

3.2.2.2.3.3 Ejecución de análisis

Los pasos para ejecutar el análisis manualmente son:

1. En el explorador de soluciones, clic en el proyecto.
2. En el menú **Analyze**, clic en **Calculate Code Metrics** on Solution.
3. El resultado es mostrado como la figura n.° 18.



Code Metrics Results

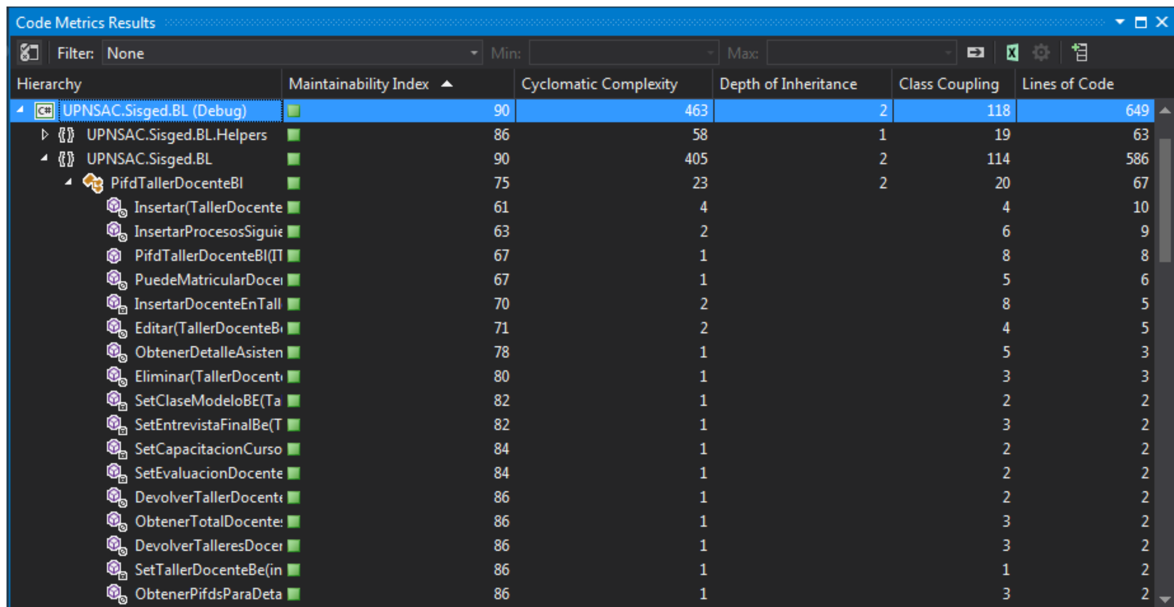
Filter: None Min: Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
One or more projects were skipp					
UPNSAC.Sisged.DAO (Debug)	72	857	3	352	3,579
UPNSAC.Sisged.WebApp (Debuc	86	1,996	4	336	2,832
UPNSAC.Sisged.BL (Debug)	90	463	2	118	649
UPNSAC.Sisged.BE (Debug)	93	968	1	71	1,025

Figura n.° 18. Resultados Visual Studio Code Metrics

Fuente: Elaboración propia

Para un análisis detallado podemos expandir los resultados por proyecto, clase y método; tal cual se visualiza en la figura n.° 19:



Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.BL (Debug)	90	463	2	118	649
UPNSAC.Sisged.BL.Helpers	86	58	1	19	63
UPNSAC.Sisged.BL	90	405	2	114	586
PifdTallerDocenteBl	75	23	2	20	67
Insertar(TallerDocente	61	4		4	10
InsertarProcesosSiguie	63	2		6	9
PifdTallerDocenteBl(IT	67	1		8	8
PuedeMatricularDoce	67	1		5	6
InsertarDocenteEnTall	70	2		8	5
Editar(TallerDocenteB	71	2		4	5
ObtenerDetalleAsisten	78	1		5	3
Eliminar(TallerDocent	80	1		3	3
SetClaseModeloBE(Ta	82	1		2	2
SetEntrevistaFinalBe(T	82	1		3	2
SetCapacitacionCurso	84	1		2	2
SetEvaluacionDocente	84	1		2	2
DevolverTallerDocent	86	1		2	2
ObtenerTotalDocente:	86	1		3	2
DevolverTalleresDocer	86	1		3	2
SetTallerDocenteBe(in	86	1		1	2
ObtenerPifdsParaDeta	86	1		3	2

Figura n.° 19. Resultados de Visual Studio Code Metrics por proyecto y clase

Fuente: Elaboración propia

3.2.2.2.4 Cloc

Es una herramienta que cuenta líneas en blanco, líneas de comentarios y líneas físicas de código fuente en muchos lenguajes de programación. Teniendo en cuenta dos versiones de una base de código, Cloc puede calcular diferencias en líneas de blanco, comentarios y líneas de origen (Wheeler, 2017).

3.2.2.2.4.1 Instalación y configuración

En sistemas Windows, Cloc se puede usar de dos maneras, una mediante su instalación para lo cual se requiere tener instalado Perl y otra simplemente descargando el ejecutable, el cual no requiere dependencias.

Los pasos para utilizar el ejecutable son los siguientes:

1. Descargar el comprimido de la última versión de Cloc desde la url http://sourceforge.net/projects/cloc/files/cloc/v1.64/.
2. Extraer el ejecutable y ubicarlo en la ruta principal del proyecto.

3.2.2.2.4.2 Ejecución de análisis

Los pasos para ejecutar el análisis manualmente son:

1. Ubicarse en la ruta principal del proyecto.
2. Abrir la consola de Windows en dicha ruta.
3. Ejecutar el comando **cloc-1.64.exe .**
4. Los resultados son mostrados por lenguaje de programación, tal cual se visualiza en la figura n. ° 20.

```
D:\Projects\sisged>cloc-1.64.exe .
    1581 text files.
     917 unique files.
    1346 files ignored.

http://cloc.sourceforge.net v 1.64  T=10.93 s (75.7 files/s, 59161.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
XML                             65           1232             56         497150
Javascript                      66          11836          14484          58487
CSS                             46           2599            723          22961
C#                             369           3650            261          19202
Razor                          132            480             75           4627
JSON                             4              0              0           2271
SQL                             93            275             28           1615
MSBuild script                   7              0             48           1256
ASP.Net                         19             26              0            621
XSLT                             1              2              7            323
PowerShell                      16             91            650            321
XSD                              1             51              3            265
LESS                             1             43              9            209
SASS                             1              3              0            196
HTML                             1             20              0            131
DOS Batch                        4             18              2             43
Puppet                           1              4              0             17
-------------------------------------------------------------------------------
SUM:                           827          20330          16346         609695
-------------------------------------------------------------------------------
```

Figura n.° 20. Resultados Cloc

Fuente: Elaboración propia

3.2.2.3. Ciclos de medición

Con el uso de la metodología Kanban se realiza el desarrollo de las mejoras en el código fuente del sistema SISGED. Las mejoras se aplican por cada indicador de calidad, los cuales son abarcados a continuación.

3.2.2.3.1 Pre test

La primera medición que se realizó corresponde al pre test, la cual nos arrojó los resultados que se aprecian a continuación:

3.2.2.3.1.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición pre test.

Tabla n.° 11. Medición pre test, herramienta Designite

Indicador	Resultado
Total #Lines of Code	17 104
Total # Clases	374
Total #Metric Violations	70
Total Code Duplication	0,25
Total # Namespaces	65
Total # Methods	1 326
Total Smell Density	18,07
Abstraction mells	239
Encapsulation smells	2
Moddularization smells	41
Hierarchy smells	27

Fuente: Elaboración propia

3.2.2.3.1.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo de medición pre test.

Tabla n.° 12. Medición pre test, herramienta SonarQube

Indicador	Resultado
Errores	974
Vulnerabilidades	21
Deuda técnica (d)	194
Olores de código	4 400
Cobertura	0,00%
Duplicados	76%
Bloques duplicados	535

Fuente: Elaboración propia

3.2.2.3.1.3 Code Metrics

Resultados de indicadores obtenidos por la herramienta Visual Studio Code Metrics, correspondientes al ciclo de medición pre test.

Tabla n.° 13. Medición pre test, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.DAO	72	784	3	345	3446
UPNSAC.Sisged.WebApp	86	1 924	4	310	2 726
UPNSAC.Sisged.BL	92	563	2	119	695
UPNSAC.Sisged.BE	93	979	1	70	1 036
UPNSAC.Sisged	86	4 250	3	844	7 903

Fuente: Elaboración propia

3.2.2.3.1.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición pre test.

Tabla n.° 14. Medición pre test, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	**51**	**1 070**	**56**	**427 823**
Javascript	**73**	**12 927**	**15 676**	**61 869**
SQL	**119**	**13 434**	**384**	**37 410**
CSS	**47**	**3 611**	**748**	**28 734**
c#	**309**	**2 733**	**226**	**14 140**
HTML	**1**	**20**	**0**	**131**

Fuente: Elaboración propia

3.2.2.3.2 Ciclo 1

El ciclo 1 nos arrojó los resultados que se aprecian en las siguientes tablas:

3.2.2.3.2.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición 1.

Tabla n.° 15. Medición ciclo 1, herramienta Designite

Indicador	Resultado
Total #Lines of Code	17 104
Total # Clases	374

Indicador	Resultado
Total #Metric Violations	70
Total Code Duplication	0,25
Total # Namespaces	65
Total # Methods	1 326
Total Smell Density	18,07
Abstraction mells	239
Encapsulation smells	2
Moddularization smells	41
Hierarchy smells	27

Fuente: Elaboración propia

3.2.2.3.2.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo 1.

Tabla n.° 16. Medición ciclo 1, herramienta SonarQube

Indicador	Resultado
Errores	1 000
Vulnerabilidades	28
Deuda técnica (d)	188
Olores de código	4 300
Cobertura	0.00%
Duplicados	86%
Bloques duplicados	926

Fuente: Elaboración propia

3.2.2.3.2.3 Code Metrics

Tabla n.° 17. Medición ciclo 1, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.DAO	72	784	3	345	3 446
UPNSAC.Sisged.WebApp	86	1 924	4	310	2 726
UPNSAC.Sisged.BL	92	563	2	119	695

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.BE	93	979	1	70	1 036
UPNSAC.Sisged	86	4 250	3	844	7 903

Fuente: Elaboración propia

3.2.2.3.2.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición 1.

Tabla n.° 18. Medición ciclo 1, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	54	1 070	56	428 017
Javascript	71	12 640	15 135	61 261
SQL	119	13 434	384	37 410
CSS	47	3 611	748	28 734
c#	309	2 733	226	14 140
HTML	1	20	0	131

Fuente: Elaboración propia

3.2.2.3.3 Ciclo 2

A partir del ciclo 2 de medición, la herramienta Designite quedó en desuso debido a la incompatibilidad con MSBuild.

El ciclo 2 de medición nos arrojó los resultados que se aprecian en las siguientes tablas:

3.2.2.3.3.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición 2.

Tabla n.° 19. Medición ciclo 2, herramienta Designite

Indicador	Resultado
Total #Lines of Code	20 702
Total # Clases	364
Total #Metric Violations	N/A

Indicador	Resultado
Total Code Duplication	N/A
Total # Namespaces	N/A
Total # Methods	N/A
Total Smell Density	N/A
Abstraction mells	N/A
Encapsulation smells	N/A
Moddularization smells	N/A
Hierarchy smells	N/A

Fuente: Elaboración propia

3.2.2.3.3.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo de medición 2.

Tabla n.° 20. Medición ciclo 2, herramienta SonarQube

Indicador	Resultado
Errores	65
Vulnerabilidades	3
Deuda técnica (d)	8
Olores de código	343
Cobertura	14,00%
Duplicados	1,3%
Bloques duplicados	18

Fuente: Elaboración propia

3.2.2.3.3.3 Code Metrics

Resultados de indicadores obtenidos por la herramienta Visual Studio Code Metrics, correspondientes al ciclo de medición 2.

Tabla n.° 21. Medición ciclo 2, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.DAO	72	787	3	350	3 412

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.WebApp	86	1 922	4	331	2 727
UPNSAC.Sisged.BL	90	463	2	120	648
UPNSAC.Sisged.BE	93	979	1	71	1 036
UPNSAC.Sisged	85	4 151	3	872	7 823

Fuente: Elaboración propia

3.2.2.3.3.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición 2.

Tabla n.° 22. Medición ciclo 2, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	64	1 232	57	476 424
Javascript	63	7 899	11299	39 314
SQL	93	275	27	1 613
CSS	46	2 592	723	22 935
c#	340	3 140	263	16 486
HTML	1	3	0	196

Fuente: Elaboración propia

3.2.2.3.4 Ciclo 3

El ciclo 3 de medición nos arrojó los resultados que se aprecian en las siguientes tablas:

3.2.2.3.4.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición 3.

Tabla n.° 23. Medición ciclo 3, herramienta Designite

Indicador	Resultado
Total #Lines of Code	20 633
Total # Clases	364

Indicador	Resultado
Total #Metric Violations	N/A
Total Code Duplication	N/A
Total # Namespaces	N/A
Total # Methods	N/A
Total Smell Density	N/A
Abstraction mells	N/A
Encapsulation smells	N/A
Moddularization smells	N/A
Hierarchy smells	N/A

Fuente: Elaboración propia

3.2.2.3.4.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo de medición 3.

Tabla n.° 24. Medición ciclo 3, herramienta SonarQube

Indicador	Resultado
Errores	65
Vulnerabilidades	3
Deuda técnica (d)	8
Olores de código	341
Cobertura	30,00%
Duplicados	1,1%
Bloques duplicados	16

Fuente: Elaboración propia

3.2.2.3.4.3 Code Metrics

Resultados de indicadores obtenidos por la herramienta Visual Studio Code Metrics, correspondientes al ciclo de medición 3.

Tabla n.° 25. Medición ciclo 3, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.DAO	72	787	3	350	3 412
UPNSAC.Sisged.WebApp	86	1 922	4	331	2 727
UPNSAC.Sisged.BL	90	461	2	119	647
UPNSAC.Sisged.BE	93	979	1	71	1 036
UPNSAC.Sisged	85	4 149	3	871	7 822

Fuente: Elaboración propia

3.2.2.3.4.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición 3.

Tabla n.° 26. Medición ciclo 3, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	64	1 232	57	477 198
Javascript	63	7 899	11 299	39 314
SQL	93	275	27	1 613
CSS	46	2 592	723	22 935
c#	343	3 202	257	16 987
HTML	1	20	0	131

Fuente: Elaboración propia

3.2.2.3.5 Ciclo 4

El ciclo 4 de medición nos arrojó los resultados que se aprecian en las siguientes tablas:

3.2.2.3.5.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición 4.

Tabla n.° 27. Medición ciclo 4, herramienta Designite

Indicador	Resultado
Total #Lines of Code	N/A

Indicador	Resultado
Total # Clases	N/A
Total #Metric Violations	N/A
Total Code Duplication	N/A
Total # Namespaces	N/A
Total # Methods	N/A
Total Smell Density	N/A
Abstraction mells	N/A
Encapsulation smells	N/A
Moddularization smells	N/A
Hierarchy smells	N/A

Fuente: Elaboración propia

3.2.2.3.5.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo de medición 4.

Tabla n.° 28. Medición ciclo 4, herramienta SonarQube

Indicador	Resultado
Errores	194
Vulnerabilidades	4
Deuda técnica (d)	39
Olores de código	790
Cobertura	29,23%
Duplicados	0,9%
Bloques duplicados	24

Fuente: Elaboración propia

3.2.2.3.5.3 Code Metrics

Resultados de indicadores obtenidos por la herramienta Visual Studio Code Metrics, correspondientes al ciclo de medición 4.

Tabla n.° 29. Medición ciclo 4, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code

UPNSAC.Sisged.DAO	72	834	3	353	3 557
UPNSAC.Sisged.WebApp	86	1 939	4	335	2 761
UPNSAC.Sisged.BL	90	463	2	118	649
UPNSAC.Sisged.BE	93	966	1	71	1 023
UPNSAC.Sisged	85	4 202	3	877	7 990

Fuente: Elaboración propia

3.2.2.3.5.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición 4.

Tabla n.° 30. Medición ciclo 4, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	63	1232	56	485 504
Javascript	65	11 815	14 308	57 700
SQL	93	275	28	1 614
CSS	47	2 617	723	23 080
c#	343	3 263	268	17 227
HTML	1	20	0	131

Fuente: Elaboración propia

3.2.2.3.6 Ciclo 5

El ciclo 5 de medición nos arrojó los resultados que se aprecian en las siguientes tablas:

3.2.2.3.6.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición 5.

Tabla n.° 31. Medición ciclo 5, herramienta Designite

Indicador	Resultado
Total #Lines of Code	N/A
Total # Clases	N/A
Total #Metric Violations	N/A
Total Code Duplication	N/A

Indicador	Resultado
Total # Namespaces	N/A
Total # Methods	N/A
Total Smell Density	N/A
Abstraction mells	N/A
Encapsulation smells	N/A
Moddularization smells	N/A
Hierarchy smells	N/A

Fuente: Elaboración propia

3.2.2.3.6.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo de medición 5.

Tabla n.° 32. Medición ciclo 5, herramienta SonarQube

Indicador	Resultado
Errores	221
Vulnerabilidades	4
Deuda técnica (d)	39
Olores de código	787
Cobertura	88,30%
Duplicados	0,0%
Bloques duplicados	0

Fuente: Elaboración propia

3.2.2.3.6.3 Code Metrics

Resultados de indicadores obtenidos por la herramienta Visual Studio Code Metrics, correspondientes al ciclo de medición 5.

Tabla n.° 33. Medición ciclo 5, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.DAO	72	858	3	354	3 587
UPNSAC.Sisged.WebApp	86	1 939	4	335	2 761

UPNSAC.Sisged.BL	90	463	2	118	649
UPNSAC.Sisged.BE	93	966	1	71	1 023
UPNSAC.Sisged	85	4 226	3	878	8 020

Fuente: Elaboración propia

3.2.2.3.6.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición 5.

Tabla n.° 34. Medición ciclo 5, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	61	1 232	56	481 310
Javascript	65	11 815	14 308	57 700
SQL	92	225	25	1 182
CSS	47	2 617	723	23 080
c#	370	3 630	263	19 045
HTML	1	20	0	131

Fuente: Elaboración propia

3.2.2.3.7 Post test

La medición final corresponde al post test, el cual arrojó los resultados que se aprecian en las siguientes tablas:

3.2.2.3.7.1 Designite

Resultados de indicadores obtenidos por la herramienta Designite, correspondientes al ciclo de medición post test.

Tabla n.° 35. Medición post test, herramienta Designite

Indicador	Resultado
Total #Lines of Code	20 173
Total # Clases	N/A
Total #Metric Violations	N/A
Total Code Duplication	N/A
Total # Namespaces	N/A

Indicador	Resultado
Total # Methods	N/A
Total Smell Density	N/A
Abstraction mells	N/A
Encapsulation smells	N/A
Moddularization smells	N/A
Hierarchy smells	N/A

Fuente: Elaboración propia

3.2.2.3.7.2 SonarQube

Resultados de indicadores obtenidos por la herramienta SonarQube, correspondientes al ciclo de medición post test.

Tabla n.° 36. Medición post test, herramienta SonarQube

Indicador	Resultado
Errores	97
Vulnerabilidades	3
Deuda técnica (d)	8
Olores de código	340
Cobertura	95,00%
Duplicados	0%
Bloques duplicados	0

Fuente: Elaboración propia

3.2.2.3.7.3 Code Metrics

Resultados de indicadores obtenidos por la herramienta Visual Studio Code Metrics, correspondientes al ciclo de medición post test.

Tabla n.° 37. Medición post test, herramienta Code Metrics

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.DAO	72	857	3	352	3 578
UPNSAC.Sisged.WebApp	86	1 939	4	335	2 761

Hierarchy	Maintainability index	Cyclomatic complexity	Depth of Inheritance	Class Coupling	Lines of Code
UPNSAC.Sisged.BL	90	463	2	118	649
UPNSAC.Sisged.BE	93	966	1	71	1 023
UPNSAC.Sisged	85	4 225	3	876	8 011

Fuente: Elaboración propia

3.2.2.3.7.4 Cloc

Resultados de indicadores obtenidos por la herramienta Cloc, correspondientes al ciclo de medición post test.

Tabla n.° 38. Medición post test, herramienta Cloc

Language	Files	Blank	Comment	Code
XML	61	1 232	56	48 118
Javascript	66	11 805	14 265	57 600
SQL	92	225	25	1 182
CSS	47	2 617	723	23 080
c#	369	3 621	261	19 083
HTML	1	20	0	131

Fuente: Elaboración propia

3.2.2.3.8 Resultado

Adicionalmente se tiene un compacto de los resultados obtenidos en cada una de las mediciones realizadas con las herramientas de calidad.

En la tabla n. ° 39, podemos observar cómo ha ido evolucionando los resultados de los indicadores en cada uno de los ciclos de medición. Considerar que a partir del ciclo 2, la herramienta Designite quedó en desuso por temas de compatibilidad con la versión de msbuild.

Tabla n.° 39. Compacto de resultados obtenidos en los ciclos de medición

Herramienta / # Ciclo	Pre test	1	2	3	4	5	Post
SonarQube							
Errores	974	1 000	65	65	194	97	97
Vulnerabilidades	21	28	3	3	4	3	3
Deuda técnica	194	188	8	8	39	8	8
Olores de código	4 400	4 300	343	341	790	343	340
Cobertura	0,00%	0,00%	14,00%	30,00%	29,23%	88,30%	95,00%
Líneas Duplicadas	76,00%	86,40%	1,3%	1,10%	0,9%	0,00%	0,00%
Bloques duplicados	535	926	18	16	24	0	0
Designite							
Total #Lines of Code	17 104	17 104	N/A	N/A	N/A	N/A	N/A
Total # Clases	374	374	N/A	N/A	N/A	N/A	N/A
Total #Metric Violations	70	70	N/A	N/A	N/A	N/A	N/A
Total Code Duplication	0,25	0,25	N/A	N/A	N/A	N/A	N/A
Total # Namespaces	65	65	N/A	N/A	N/A	N/A	N/A
Total # Methods	1 326	1 326	N/A	N/A	N/A	N/A	N/A
Total Smell Density	18,07	18,07	N/A	N/A	N/A	N/A	N/A

Herramienta / # Ciclo	Pre test	1	2	3	4	5	Post
Abstraction mells	239	239	N/A	N/A	N/A	N/A	N/A
Encapsulation smells	2	2	N/A	N/A	N/A	N/A	N/A
Hierarchy smells	27	27	N/A	N/A	N/A	N/A	N/A
Code Metrics							
Maintainability index	86	86	85	85	85	85	85
Cyclomatic complexity	4 250	4 250	4 151	4 149	4 202	4 226	4 225
Depth of Inheritance	3	3	3	3	3	3	3
Class Coupling	844	844	872	871	877	878	876
Lines of Code	7 903	7 903	7 823	78 22	7 990	8 020	8 011
CLOC							
Javascript	61 869	61 261	39 314	39 314	57 700	57 700	57 600
SQL	37 410	37 410	1 613	1 613	1 614	1 182	1 182
c#	14 140	14 140	16 486	16 987	17 227	19 045	19 083
Total LOC	113 419	112 811	57 413	57 914	76 541	77 927	77 865
Líneas en Blanco	2 733	2 733	3 140	3 202	3 263	3 630	3 621
Comentarios	226	226	263	257	268	263	261

Fuente: Elaboración propia

3.2.3. Metodología de investigación científica

Se está trabajando con la metodología de investigación científica ya que está validada, probada y verificada por expertos.

3.2.3.1. ¿Por qué la metodología de investigación científica?

Se eligió la metodología de investigación científica, por la estructura y naturaleza de la investigación.

3.2.3.2. Desarrollo del método científico.

El método científico se encuentra implícito en todo el desarrollo del informe.

3.3. Nivel de investigación

El nivel de investigación es correlacional ya que se pretende medir el grado de relación entre la variable independiente y la variable dependiente.

3.4. Diseño de investigación

Es una investigación experimental: Quasi-experimental, ya que se manipuló de manera intencional las variables independientes para analizar las consecuencias sobre las variables dependientes.

Diseño Experimental:

Tabla n.° 40. Diseño experimental

Grupo	Asignación	Pre Prueba	Tratamiento	Post Prueba
GE	=>	O1	X	O2

Fuente: Elaboración propia

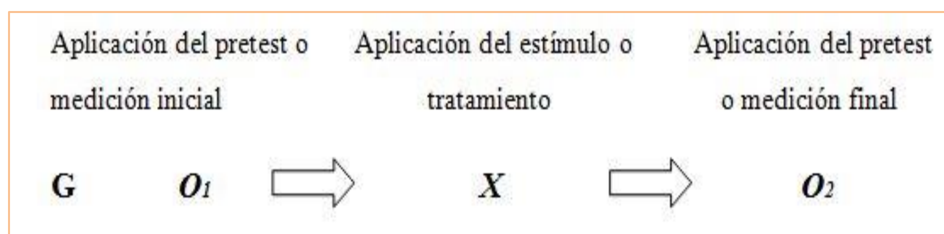


Figura n.° 21. Diseño quasi-experimental

Fuente: Elaboración propia

Dónde:

GE: Grupo de estudio

O1: Pre test

O2: Post test

3.4.1. Grupos de estudio

Tabla n.° 41. Subdivisión de grupos de estudio

Ge	Pre Prueba	Tratamiento	Post Prueba
Errores	974	Mejora de código	97
Vulnerabilidades	21	Mejora de código	3
Deuda técnica	194	Creación de pruebas unitarias	8
Olores de código	4400	Mejora de código	340
Líneas duplicadas	76%	Mejora de código	0%
Bloques duplicados	535	Mejora de código	0
Complejidad ciclomática	4250	Mejora de código	4225
Índice de mantenimiento	86	Mejora de código	85
Dependencia de herencia	3		3
Líneas de código	7903	Crecimiento automático	8011
Líneas en blanco	2733		3621
Comentarios	226		261

Fuente: Elaboración propia

3.5. Unidad de estudio

Cada línea de código del sistema SISGED.

3.6. Población

Para la presente investigación se considera como población al código fuente del sistema SISGED.

El tamaño de la población equivale al total de líneas de código del sistema SISGED, que entre la suma de todos sus módulos hacen un total de **45244**.

Tabla n.° 42. Total de líneas de código

Lenguaje	Total
C#	14079
Javascript	31165

Fuente: Elaboración propia

Dado el estímulo o tratamiento, se incrementó el total de líneas de código del sistema SISGED.

3.7. Muestra

Debido a que no se analizará una parte de código o algún módulo en específico, la muestra será del tipo poblacional por analizarse todo el código fuente del sistema SISGED.

El tamaño de la muestra equivale al tamaño total de la población, 45244 líneas de código.

3.8. Técnicas, instrumentos y procedimientos de recolección de datos

Cómo técnicas e instrumentos de recolección de datos se utilizarán las siguientes:

3.8.1. Técnicas – Métodos

- Herramientas de calidad para la evaluación de métricas.
- Fichas de resultados de indicadores.

3.8.2. Instrumentos

- Herramientas de calidad.
- Ficha de resultados de indicadores SonarQube (ver ficha en anexo n.° 1)
- Ficha de resultados de indicadores Designite (ver ficha en anexo n.° 2)
- Ficha de resultados de indicadores Cloc (ver ficha en anexo n.° 3)
- Ficha de resultados de indicadores Visual Studio Code Metrics (ver ficha en anexo n.° 4)

3.9. Métodos, instrumentos y procedimientos de análisis de datos

Cómo métodos y procedimientos de análisis de datos se utilizarán los siguientes:

3.9.1. Métodos

1. Software estadístico **SPSS 22.0.0.0**
2. Prueba de **chi-cuadrado** para determinar si existe una relación entre las dos variables mencionadas en apartados anteriores.

3.9.2. Procedimientos y herramientas

El procedimiento a seguir será el análisis estático de código; para lo cual se seguirá el flujo:

1. Primera ejecución análisis estático de código fuente.
2. Anotación de resultados obtenidos.
3. Mejora progresiva de código fuente, según los resultados obtenidos previamente.
4. Ejecución de análisis estático de código fuente con las nuevas mejoras sobre el código.
5. Se repiten los pasos del 2 al 4 hasta alcanzar un nivel aceptable en cuanto a las métricas de calidad de código (un total de 7 ciclos).
6. Comparación de resultados obtenidos en todos los ciclos de medición de métricas.

Para el procedimiento indicado anteriormente, las herramientas a usar serán:

1. SonarQube: herramienta para medir la calidad del código (evaluación estática y de caja blanca, principalmente). Sonar obtiene métricas del código y muestra los resultados en un cuadro de mando (SonarSource S.A, 2017).
2. Visual Studio Code Metrics: herramienta que indica el nivel de complejidad y mantenimiento del código fuente dentro de una solución o proyecto de Visual Studio.
3. Cloc: herramienta que cuenta las líneas en blanco, líneas de comentarios y líneas físicas de código fuente en muchos lenguajes de programación. Dadas dos versiones de una base de código, CLOC puede calcular las diferencias en las líneas en blanco, comentarios y fuentes (Wheeler, 2017).
4. Designite: es una herramienta de evaluación de calidad de diseño de software, la cual no sólo es compatible con el diseño integral de detección de olores de código, sino que también proporciona un análisis detallado de métricas de calidad (Sharma, Mishra, & Tiwari, 2016).

3.10. Viabilidad económica del proyecto

3.10.1. Recursos humanos.

Entre los recursos humanos que participaron en la investigación, tenemos:

- Tesista
- Asesora
- Product Owner

3.10.2. Materiales.

Entre los recursos materiales que se utilizaron en la investigación, tenemos:

- Laptop
- Impresora

- Hojas bond

3.10.3. Técnicos

Entre los recursos técnicos que se utilizaron en la investigación, tenemos:

- Herramientas de análisis de código: SonarQube, Designite, Cloc, Code Analysis
- Visual Studio 2015 Express
- SQL Server 2012 Express
- Microsoft Office Word

3.10.4. Servicios

Entre los servicios que se utilizaron en la investigación, tenemos:

- Luz
- Internet
- Amazon web services
- SonarQube

3.10.5. Presupuesto

Para el presupuesto mencionado a continuación, se va a considerar 7 meses de trabajo dedicados a la realización de la investigación.

Tabla n.° 43. Presupuesto del proyecto

Tipo	Concepto	Costo Mes/Año	N° Meses	Total
Académico	Tesista: considerando un total de 280 horas en los 7 meses y a un costo de S/ 10 la hora.	S/. 400.00	7	S/. 2,800.00
	Asesora	S/. 300.00	N/A	S/. 300.00
Personal	Pasajes: considerando un costo mensual de S/. 20 en gasolina.	S/. 30.00	7	S/. 210.00
	Viáticos	S/. 28.00	7	S/. 196.00
Materiales	Laptop	S/. 220.00	N/A	S/. 1,280.00
	Internet	S/. 40.00	7	S/. 280.00

Tipo	Concepto	Costo Mes/Año	N° Meses	Total
	Luz	S/. 20.00	7	S/. 140.00
Servicios	Amazon EC2: se va a alquilar un servidor de 4GB de RAM a un costo de $0.065 por hora y con un consumo promedio de 280 horas.	S/. 60.00	7	S/. 420.00
Total				S/. 5,626.00

Fuente: Elaboración propia

3.10.6. Financiamiento

Entre las fuentes de financiamiento para la realización de la investigación, tenemos:

- Autofinanciado por el investigador.

CAPÍTULO 4. RESULTADOS

En este acápite se aborda lo concerniente a los resultados obtenidos durante el desarrollo de la investigación, donde se sacarán a relucir los hallazgos más sobresalientes de las evaluaciones aplicadas. Dichos resultados serán presentados por objetivos.

Tabla n.° 44. Estadísticos de prueba SonarQube

	Errores	Vulnerabilidades	Deuda Técnica	Olores de Código	Cobertura de Código	Líneas Duplicadas	Bloques Duplicados
Chi-cuadrado	,857[a]	3,857[b]	3,857[b]	,714[c]	,714[c]	,714[c]	,714[c]
gl	4	3	3	5	5	5	5
Sig. Asint.	,931	,277	,277	,982	,982	,982	,982

Fuente: Elaboración propia

Como el valor del significado asintótico es mayor a 0,0 5 en cada uno de los indicadores, podemos afirmar que hay una relación significativa entre las métricas analizadas por la herramienta SonarQube y el impacto en la calidad de código del sistema SISGED; por lo tanto, podemos aceptar la hipótesis propuesta.

Tabla n.° 45. Estadísticos de prueba Code Analysis

	Complejidad Ciclomática	Índice de Mantenimiento	Líneas de Código
Chi-cuadrado	,714[a]	1,286[b]	,000[c]
Gl	5	1	6
Sig. asintótica	,982	,257	1,000

Fuente: Elaboración propia

Como el valor del significado asintótico es mayor a 0,0 5 en cada uno de los indicadores, podemos afirmar que hay una relación significativa entre las métricas analizadas por la herramienta Code Analysis y el impacto en la calidad de código del sistema SISGED; por lo tanto, podemos aceptar la hipótesis propuesta.

Tabla n.° 46. Estadísticos de prueba Cloc

	Líneas en Blanco	Comentarios
Chi-cuadrado	,714[a]	,857[b]
gl	5	4
Sig. asintótica	,982	,931

Fuente: Elaboración propia

Como el valor del significado asintótico es mayor a 0,0 5 en cada uno de los indicadores, podemos afirmar que hay una relación significativa entre las métricas analizadas por la herramienta Cloc y el impacto en la calidad de código del sistema SISGED; por lo tanto, podemos aceptar la hipótesis propuesta.

En cuanto a las herramientas que ayudan en la implementación de software de calidad, estas se agrupan en tres categorías: calidad del producto software, Testing y Scrum; en cada categoría podemos encontrar una amplia variedad de herramientas que nos aseguran implementar software de calidad. Para los fines de la investigación solo se centró en la calidad del producto software y entre las herramientas que mejores resultados nos brindaron tenemos SonarQube, Visual Studio Code Analysis y Cloc; se eligió estas tres herramientas al ser de rápida configuración, por su facilidad de uso, la información detallada que muestran luego de la ejecución de un análisis, no son de pago y sobretodo son compatibles con el lenguaje de programación utilizado en el proyecto SISGED.

Para determinar el porcentaje de mejora de un producto software con la utilización de herramientas de calidad, se realizó un análisis a los resultados de los principales indicadores de calidad, estos tomados según los antecedentes mencionados al inicio de la investigación.

En la tabla n.° 47, podemos observar los niveles de impacto y porcentaje referencial de mejora a considerar:

Tabla n.° 47. Niveles de impacto

Valor	Impacto	Porcentaje referencial de mejora
-3	Alto negativo	-67% a -100%
-2	Medio negativo	-34% a -66%
-1	Bajo negativo	-1% a -33%
0	No hay impacto	0%

Valor	Impacto	Porcentaje referencial de mejora
1	Bajo positivo	1% a 33%
2	Medio positivo	34% a 66%
3	Alto positivo	67% a 100%

Fuente: Elaboración propia

Debido a que las unidades de medida de los indicadores no es la misma, se va a considerar un valor referencial.

El cálculo del porcentaje de mejora se realiza aplicando la regla de tres simple, por ejemplo: para el caso de deuda técnica se tiene:

- Pre prueba: 194 días
- Post prueba: 8 días (se redujo 186 días)
- Impacto: positivo, al evidenciarse la reducción del número de días que tomará en corregirse los errores del sistema SISGED.
- Entonces:
    - 194 -> 100%
    - 186 -> x%

El resultado de la operación nos da 95%; y este valor se ubica en el nivel de impacto **Alto positivo**.

Tabla n.° 48. Cálculo del impacto de los indicadores

Indicador	Und. Medida	Pre Prueba	Post Prueba	Mejora	Nivel De Impacto
Deuda técnica	Días	194	8	95%	3
Errores	Número	974	21	97%	3
Cobertura de código	Porcentaje	0	95	95%	3
Líneas duplicadas	Porcentaje	76	0	100%	3
Complejidad ciclomática	Número	9,9	10,5	-6%	-1
Líneas de código	Número	45 244	20 292	44,85%	2
Resultado ponderado				70,4%	3

Fuente: Elaboración propia

Por lo tanto podemos decir que el porcentaje de mejora del sistema SISGED con el uso de herramientas software de calidad fue de 70,4%.

Durante el desarrollo de las mejoras propuestas por las herramientas de calidad, se evidenció claramente que el uso de la metodología de desarrollo de software Kanban aumentó la eficacia de este proceso; esto gracias a la principal característica de Kanban que es la visibilidad (uso de tarjetas y tablero Kanban). Primeramente el uso de las tarjetas Kanban ayuda en la realización de las tareas al contener toda la información necesaria para desarrollar una mejora; en segunda instancia está el uso del tablero Kanban, que nos ayuda a tener una visión general de todo el avance que se viene realizando en la fase de desarrollo. Adicionalmente está la metodología en sí, que al ser ágil, incrementa el proceso de entrega continua de las mejoras desarrolladas.

Para determinar la influencia de la calidad interna del código sobre la calidad externa del software se utilizó la herramienta PageSpeed Insights de Google, la cual nos muestra una influencia positiva en cuanto a la velocidad de carga del sistema. Como podemos observar en la figura n.° 22, antes de realizadas las mejoras, obtenemos una puntuación de 77 sobre un total de 100 lo cual no es malo; pero luego de realizadas las mejoras en la calidad de código obtenemos una puntuación de 91, lo cual evidentemente nos indica que hubo una influencia positiva en la calidad externa del software (ver figura n.° 23).

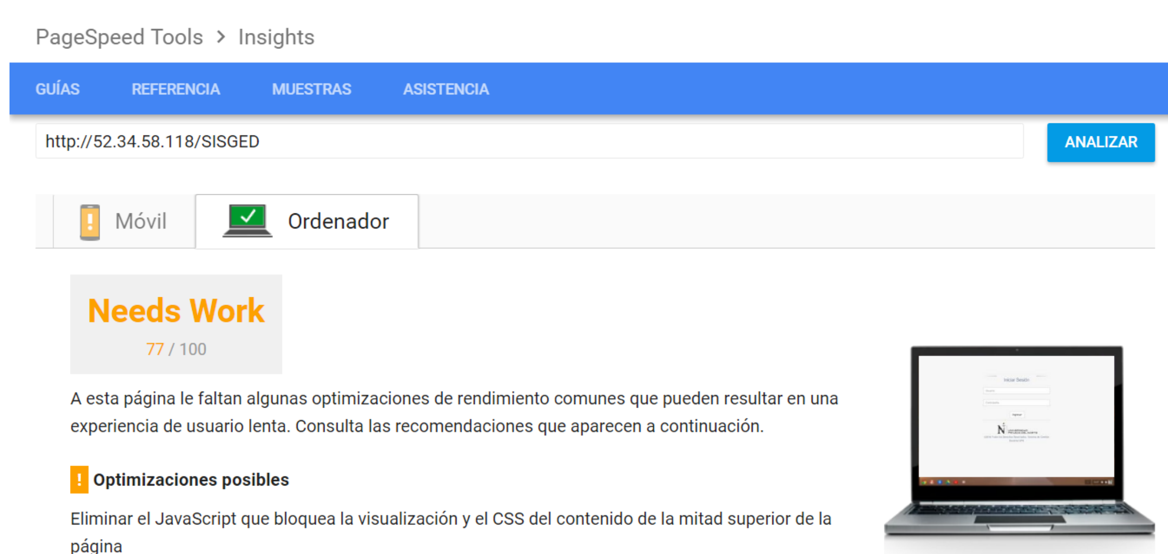


Figura n.° 22. Velocidad de carga en pre test

Fuente: (Google, 2014)

Figura n.° 23. Velocidad de carga en post test

Fuente: (Google, 2014)

Luego de realizadas las mediciones de las métricas de calidad, la herramienta que nos brinda un mayor detalle del nivel de mantenibilidad de un producto software es SonarQube. Tal cual se puede apreciar en la figura n.° 24, los niveles de mantenibilidad establecidos por (SonarSource S.A, 2017) van de la letra A hasta la letra E y para el caso del nivel de mantenibilidad del sistema SISGED se obtiene una calificación de A, estableciendo esto el mejor nivel de mantenibilidad.

Maintainability

Name	Key	Description
Code Smells	code_smells	Number of code smells.
New Code Smells	new_code_smells	Number of new code smells.
Maintainability Rating (formerly SQALE Rating)	sqale_rating	Rating given to your project related to the value of your Technical Debt Ratio. The default Maintainability Rating grid is: A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1 The Maintainability Rating scale can be alternately stated by saying that if the outstanding remediation cost is: • <=5% of the time that has already gone into the application, the rating is A • between 6 to 10% the rating is a B • between 11 to 20% the rating is a C • between 21 to 50% the rating is a D • anything over 50% is an E
Technical Debt	sqale_index	Effort to fix all maintainability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
Technical Debt on new code	new_technical_debt	Technical Debt of new code

Technical Debt Ratio	sqale_debt_ratio	Ratio between the cost to develop the software and the cost to fix it. The Technical Debt Ratio formula is:
		`Remediation cost / Development cost`
		Which can be restated as:
		`Remediation cost / (Cost to develop 1 line of code * Number of lines of code)`
		The value of the cost to develop a line of code is 0.06 days.

Technical Debt Ratio on new code	new_sqale_debt_ratio	Ratio between the cost to develop the code changed in the leak period and the cost of the issues linked to it.

Figura n.° 24. Definición de métricas de Mantenibilidad de SonarQube

Fuente: (SonarSource S.A, 2017)

Se determinó el nivel de mejora con el uso de herramientas de calidad a cada métrica analizada, como podemos observar en las figuras n.° 25 al n.° 29. En estas figuras se muestra como fue la evolución de los principales indicadores de calidad de código, para lo cual tenemos: a) La cobertura de código fue desde un nivel de 0% hasta alcanzar un máximo de 95%, b) El porcentaje de líneas duplicadas fue desde un nivel de 76% hasta llegar a un 0%, c) Los bloques duplicados se redujeron de 535 a un total de 0, d) La complejidad ciclomática se redujo de 4250 a un total de 4225, e) El total líneas de código disminuyo de 113419 a un total de 77865; por lo tanto podemos decir que el nivel de mejora de código fue significativo con el uso de herramientas de calidad de código.

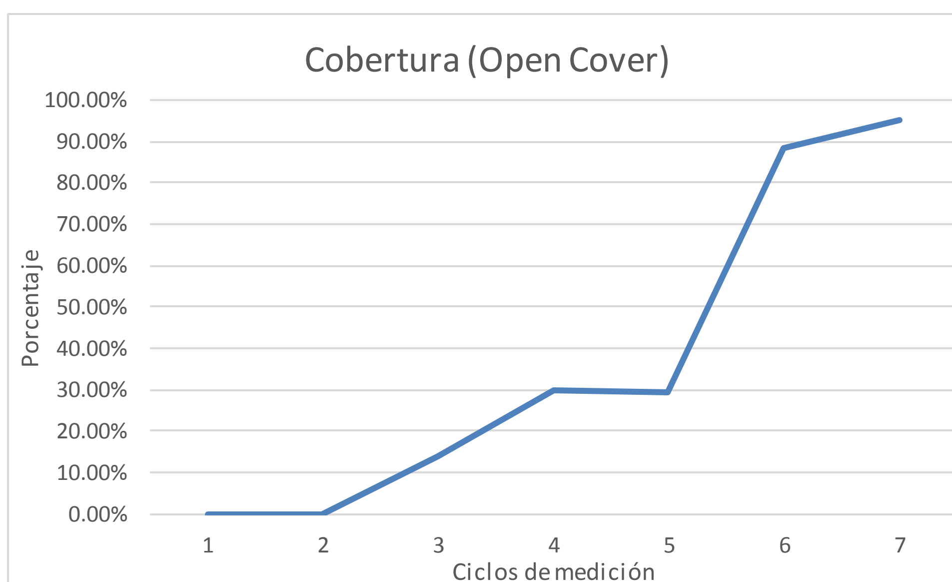


Figura n.° 25. Evolución de cobertura de código

Fuente: Elaboración propia

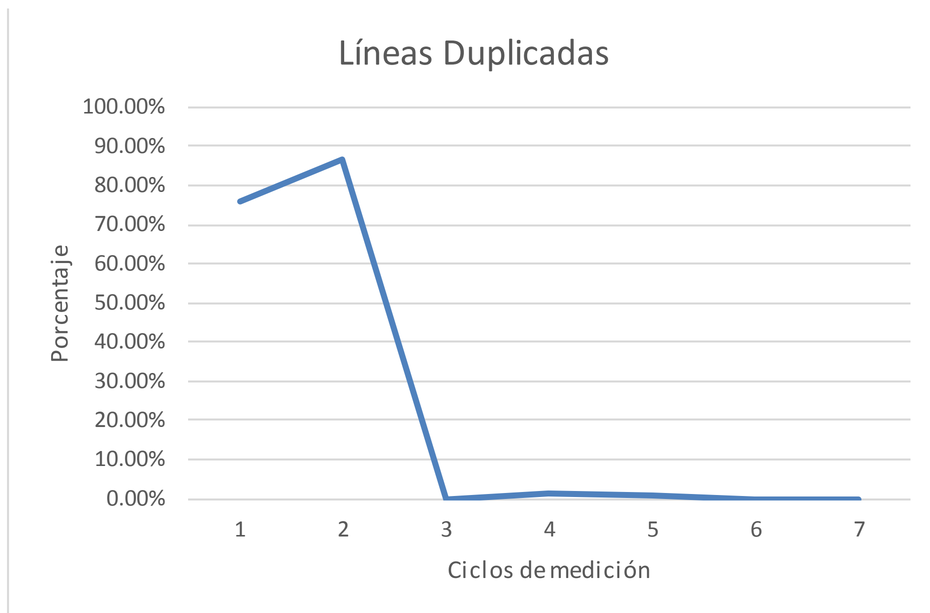


Figura n.° 26. Evolución de líneas duplicadas

Fuente: Elaboración propia

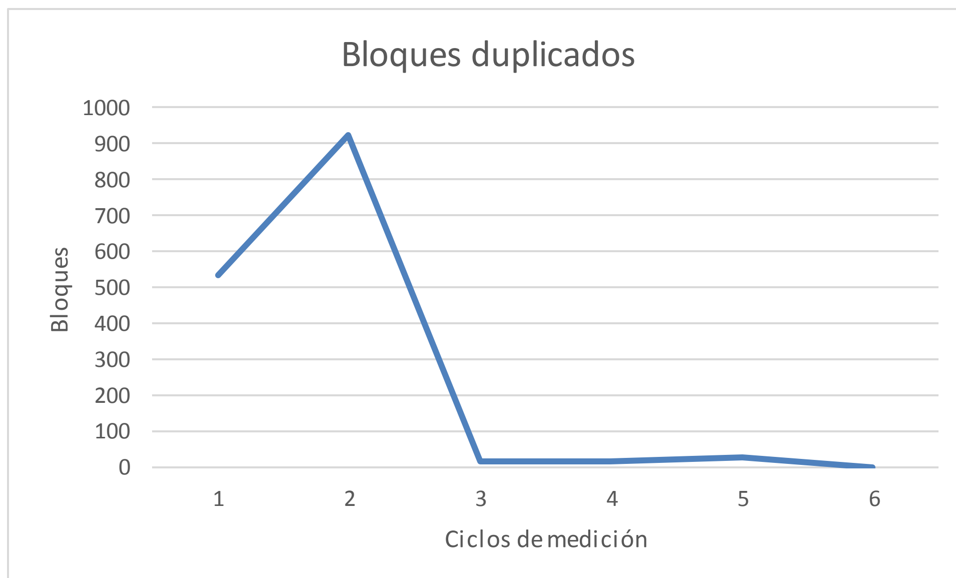


Figura n.° 27. Evolución de bloques duplicados

Fuente: Elaboración propia

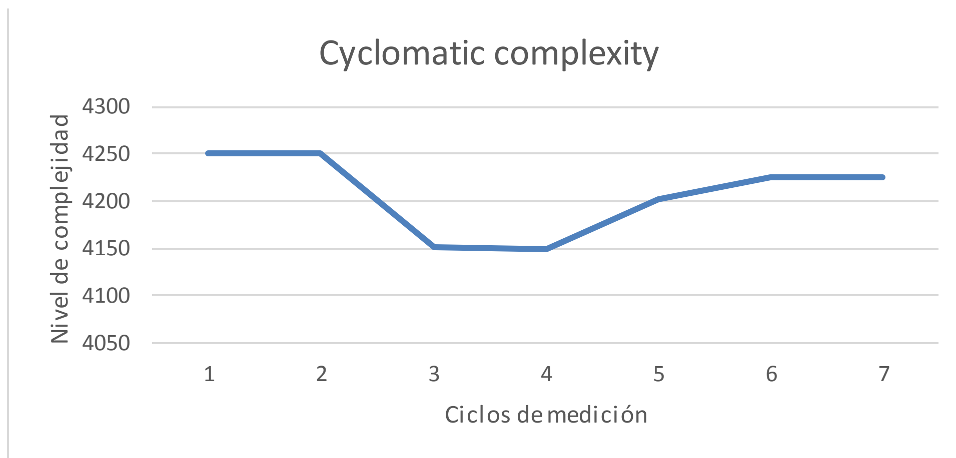


Figura n.° 28. Evolución de complejidad ciclomática

Fuente: Elaboración propia

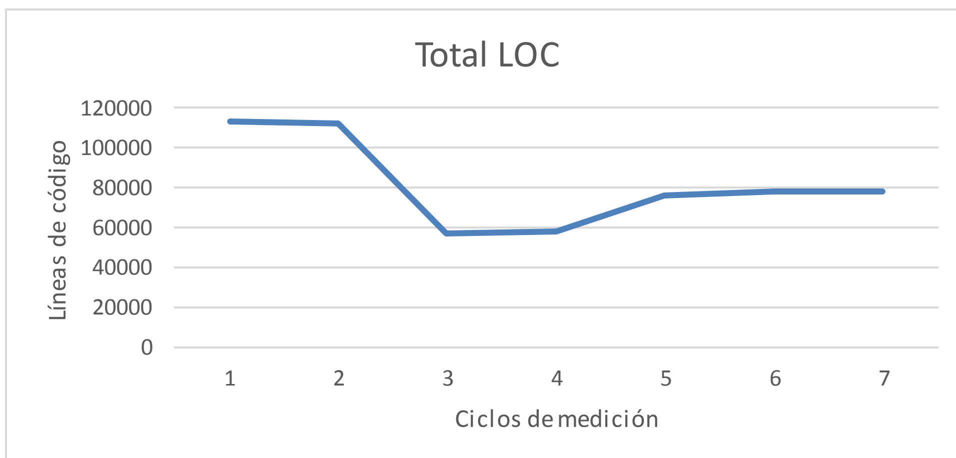


Figura n.° 29. Evolución de líneas de código

Fuente: Elaboración propia

CAPÍTULO 5. DISCUSIÓN

En esta investigación se pretendió determinar el impacto de las herramientas de calidad en la implementación de software de calidad, ya que como se indica en nuestra realidad problemática, empresas y/o profesionales que se dedican al desarrollo de software no se preocupan por garantizar un software de calidad, esto ya sea por los costes extra que implica o porque desconocen formas de lograrlo.

En nuestro estudio se emplearon herramientas que miden indicadores de calidad de código y a la vez nos proporcionan recomendaciones de como tener un código de mejor calidad. Entre los indicadores abarcados tenemos: errores, vulnerabilidades, deuda técnica, olores de código, cobertura de código, líneas duplicadas, bloques duplicados, índice de mantenibilidad y complejidad ciclomática; del mismo modo podemos comparar que en nuestro antecedente Clean Code menciona algunos de estos indicadores, los cuales son considerados para la correcta construcción de un producto software de calidad.

Durante el proceso de recolección de datos, se tuvo problemas con el uso de la herramienta Designite esto debido a que trabaja con versiones anteriores o iguales a msbuild 12.0 y en nuestro caso luego del segundo ciclo de medición de resultados, se tuvo que actualizar msbuild a la versión 14.0 por temas de compatibilidad con SonarQube, quedando esta herramienta en desuso. Pero en el tiempo que se utilizó se pudo evidenciar lo mencionado por Tushar Sharma, Pratibha Mishra y Rohit Tiwari (2016) quienes nos indican que dicha herramienta es eficaz en el análisis detallado de Métricas de Código y ayuda a mejorar la agilidad del diseño de software, por lo tanto estamos de acuerdo con los resultados presentados por dicho autores. Cabe mencionar que sería recomendable que esta herramienta soporte versiones recientes de msbuild de modo que sirva de apoyo a quienes están en constante cambio y actualización de herramientas en sus ambientes de trabajo.

También en base a nuestros resultados obtenidos podemos aceptar lo mencionado por Francesca Fontana, Ricardo Roveda y Marco Zanoni (2016), quienes indican que el uso de herramientas de calidad como SonarQube tienen un impacto positivo en la mejora de la calidad de un producto software y adicionalmente la reducción de la deuda técnica lo cual se pudo observar luego de aplicado el post test. Esto se validó tomando como base la variable independiente Métricas a Evaluar, para la cual tenemos según herramienta SonarQube, Visual Studio Code Metrics y Cloc valores de 7, 4 y 3 respectivamente y al aplicar a nuestros datos la prueba de chi-cuadrado, el valor del significado asintótico es mayor a 0,05 para los tres casos, lo cual nos indica que si existe una significancia estadística de nuestros datos, con lo cual aceptamos la conclusión de los autores.

Además podemos mencionar que no se requiere mucha experiencia para poder usar y entender las herramientas SonarQube, Visual Studio Code Metrics y Cloc; y si tomamos lo dicho por Toshisa Tanaka, Hiroshi Ogasawa, Ayako Yamada y Mitsuhiro Aizawa (2002) y en base nuestros resultados, es recomendable el uso de estas herramientas por empresas y profesionales dedicados al desarrollo de software gracias a los beneficios que trae en la mejora de calidad de código. Esto se acepta gracias a la medición de la variable independiente nivel de facilidad de uso, la cual aplicada la prueba de chi-cuadrado nos da un valor del significado asintótico mayor a 0,05, indicando esto que la facilidad de uso está presente para las tres herramientas presentadas. Esto nos hace aceptar la conclusión propuesta por los autores.

En lo que concierne a la variable porcentaje de indicadores abarcados tenemos que la herramienta SonarQube tiene un amplio rango de métricas evaluadas, resultando esto en ser la herramienta que mejor aplica para nuestro objetivo y tomando como referencia a Ryan Pinkham (2016), quien menciona que las revisiones de código son una de las mejores formas de garantizar e implementar software de calidad resultando esto en la influencia positiva sobre la calidad de software que se construye, podemos decir que hemos llegado a la misma conclusión y los resultados nos respaldan ya que luego de aplicada la prueba chi-cuadrado a esta variable obtenemos un del significado asintótico mayor a 0,05, indicando esto que SonarQube es la herramienta de mayor influencia como herramienta que asegura el software de calidad.

CAPÍTULO 6. CONCLUSIONES

Se determinó que el uso de herramientas de calidad de código tiene un impacto positivo en la implementación de un producto software de calidad y en comparación con los antecedentes que nos ratifican, podemos decir que usar estas herramientas de calidad durante el ciclo de vida del desarrollo de software garantiza la detección de olores de código, errores y vulnerabilidades; tres puntos esenciales para tener asegurado el desarrollo de un software de calidad. Entonces podemos decir que se logró cumplir con el objetivo general de la investigación.

Se utilizó como herramientas que aseguran la implementación de software de calidad SonarQube, Visual Studio Code Metrics, Cloc y Designite; estas debido a que soportan el análisis estático de código fuente en lenguaje C#; esto obviamente porque el sistema SISGED se encuentra desarrollado en dicho lenguaje, además porque estas herramientas no son de pago y son fáciles de configurar y usar. Se determinó que la herramienta que más influye en la calidad de código es SonarQube, al abarcar un mayor número de métricas de calidad y tener un alto nivel de facilidad de uso.

Se logró determinar el porcentaje de mejora del sistema SISGED, con el uso de herramientas de calidad de código, este porcentaje en base a las métricas de calidad analizadas, la cuales tenemos a) La deuda técnica mejoró un 95%, b) Los errores se redujeron un 97%, c) La cobertura de código se mejora un 95%, d) Las líneas duplicadas de redujeron un 100%, e) La complejidad ciclomática aumento un 6%, f) Las líneas de código se redujeron en un 44,85%, al final calculando el promedio de todos los indicadores, se logró mejorar un 70,4% la calidad de código del sistema SISGED.

Se logró aplicar satisfactoriamente las mejoras propuestas por las herramientas de calidad, esto gracias a la forma entendible de cómo se muestran en los cuadros de mando de dichas herramientas, la gran ventaja es que no solamente te muestran la violación de la métrica de calidad, sino también te explican porque se detecta como violación de métrica, cómo identificar dicho problema encontrado y ejemplos de posibles mejoras que puedes aplicar al código fuente. Además tenemos la metodología de desarrollo Kanban con la cual se desarrollaron estas mejoras; donde se evidenció claramente que el uso de esta metodología de desarrollo de software aumentó la eficacia de este proceso; esto gracias al uso de tarjetas y tablero Kanban.

Se logró determinar que las mejoras realizadas en la calidad interna del código tienen una influencia positiva en la calidad externa del sistema SISGED, esto gracias a la ayuda de las herramientas SonarQube y PageSpeed Insights. En el primer caso, debido a que SonarQube indica archivos que están siendo llamados por la aplicación, pero que no están siendo utilizados o bloques de código que igualmente no están siendo utilizados; lo cual evidentemente tiene un

impacto negativo en el tiempo de carga del sistema. Por otro lado tenemos a PageSpeed Insights que no arrojó un valor de 91 sobre un total de 100 a comparación del valor 77 que se tenía antes de realizadas las mejoras al sistema SISGED.

Se determinó que SonarQube es la mejor herramienta que nos ayuda a controlar el nivel de mantenibilidad de un producto software. Adicionalmente se logró reducir considerablemente el nivel de mantenibilidad del sistema SISGED, obteniendo un valor de cero en código duplicado y deuda técnica de 8 días, esto luego de realizadas las mejoras en el código fuente del sistema SISGED; logrando obtener una calificación de mantenibilidad de A entre un rango de la A hasta la E, que es el mejor nivel de mantenibilidad según (SonarSource S.A, 2017). Si esto lo traducimos a términos económicos, quiere decir que solo requerimos de 8 días para corregir todos los olores de código, errores y vulnerabilidades existentes en el sistema SISGED.

Se logró implementar en el código fuente del sistema SISGED, gran parte de las mejoras propuestas por las herramientas de calidad de código, consiguiendo:

- Reducir la deuda técnica de 194 días a 8 días.
- Aumentar la cobertura de código un 95%.
- Reducir los errores y vulnerabilidades de 974 a 97 y de 21 a 3 respectivamente.
- Reducir las líneas duplicadas de un 76% a 0%.
- Reducir la complejidad ciclomática al rango de métodos sencillos definidos por Thomas McCabe: 10.5.
- Reducir el total de líneas de código un 44,85%.

CAPÍTULO 7. RECOMENDACIONES

Se recomienda, no solo a empresas desarrolladoras de software, sino a todo profesional y estudiantes utilizar herramientas de calidad de software en sus proyectos de desarrollo, ya que como se observó en los resultados la influencia es positiva en la construcción de software de calidad.

Asimismo, de las herramientas utilizadas en la investigación, se recomienda utilizar SonarQube por el apoyo visual que nos brinda su Dashboard en cuanto a las métricas de calidad evaluadas, es libre de pago, de fácil uso e instalación y además porque nos ayuda a obtener métricas que aportan en mejorar la calidad de un producto software.

A todo aquel que desarrolle proyectos de software en el IDE Visual Studio, sería recomendable utilizar la herramienta Code Metrics para verificar si los valores de la complejidad ciclomática e índice de mantenibilidad se encuentran entre el rango permitido.

Para futuras investigaciones, estudiantes universitarios interesados en el tema, podrían dar más énfasis al significado de los indicadores mostrados por la herramienta Visual Studio Code Metrics ya que en la presente investigación se enfatizó solo los indicadores mostrados en SonarQube.

Como se mencionó en apartados anteriores, existen 6 características de calidad del software, de las cuales solo se abarcó la mantenibilidad; por ende estudiantes universitarios pueden continuar investigando el impacto en la calidad final de un producto software con el uso de herramientas para cada una de estas características.

Recomendaría una segunda investigación, para determinar si:

- Incluir las herramientas de calidad durante el proceso de desarrollo del software tiene algún tipo de impacto respecto al costo, alcance y tiempo del proyecto.
- Cuanto incrementaría en costos, incluir las herramientas de calidad.

Para mejorar la calidad del software que se construye, los estudiantes y profesionales deberían estudiar el libro Clean Code (Martin, 2011) y aplicar todas las buenas prácticas de programación ahí mencionadas.

CAPÍTULO 8. REFERENCIAS

Aldazabal Gil, L. (26 de Marzo de 2015). *Code2Read*. Obtenido de https://code2read.com/2015/03/25/code-metrics-complejidad-ciclomatica/

Angeles Estrada, J. (Junio de 2006). *Sistema Kanban, como una ventaja competitiva en la Micro, Pequeña y Mediana Empresa.* México.

Atlassian. (2017). *Atlassian*. Obtenido de https://es.atlassian.com/agile/kanban

Bowes, J. (21 de Julio de 2015). *MANIFESTO*. Obtenido de https://manifesto.co.uk/kanban-vs-scrum-vs-xp-an-agile-comparison/

Bubevski, V. (2013). A novel approach to software quality risk management. *Wilwy Online Library*, 154.

Cruz, R. (2010). *Asegurar la calidad del código, un primer paso hacia la mejora de la calidad global del software.* Madrid.

Cunningham, W. (2001). *Manifesto for Agile Software Development*. Obtenido de Manifesto for Agile Software Development: http://agilemanifesto.org/

Dirección General de Servicio Civil. (2013). *Modelo de Calidad de Software pra Desarrollo de Sistemas en la DGSC.*

Fontana, F., Roveda, R., & Zanoni, M. (2016). Technical Debt Indexes Provided by Tools: A Preliminary Discussion. *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)* (pág. 42). IEEE.

Fowler, M. (1999). *Refactoring:Improving the Design of Existing Code.* ADDISON WESLEY LONGMAN INC.

Garzas, J. (22 de Noviembre de 2012). *javiergarzas.com*. Obtenido de javiergarzas.com: http://www.javiergarzas.com/2012/11/deuda-tecnica-2.html

Ghahrai, A. (21 de 05 de 2016). *Testing Excellence*. Obtenido de https://www.testingexcellence.com/difference-between-scrum-kanban-xp-agile/

Google. (11 de Abril de 2014). *Google Developers*. Obtenido de https://developers.google.com/speed/docs/insights/about?hl=es-ES&utm_source=PSI&utm_medium=incoming-link&utm_campaign=PSI

IEEE. (2010). *IEEE Advancing Technology for Humanity*. Obtenido de https://www.ieee.org/index.html

International Organization for Standardization. (15 de Julio de 1995). Quality management and quality assurance.

International Organization for Standardization. (2015). *Quality management systems -- Requirements*. Obtenido de ISO 9001:2008: https://www.iso.org/standard/46486.html

Juran, J. (1992). *Departmental Quality Planning.* Departmental quality planning.

Kulikova, I. (07 de Octubre de 2016). *Project Management*. Obtenido de https://www.projectmanagement.com/blog-post/23006/Scrum-vs-Kanban-vs-XP

Martin, R. C. (2011). *Clen Code.* FINANCIAL TIMES/PRENTICE HALL.

Microsoft. (2017). *Microsoft Developer Network*. Obtenido de https://msdn.microsoft.com/en-us

Mills, E. (1998). Software Metrics. *SEI Curriculum Module SEI-CM-12-1.1*, 43.

Ospina, J. (2015). *Análisis de seguridad y calidad de aplicaciones.* Manizales.

Oxford University Press. (27 de Febrero de 2004). *Encyclopedia.com.* Obtenido de http://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/software-tool

Pinkham, R. (28 de Marzo de 2016). *SMARTBEAR*. Obtenido de http://blog.smartbear.com/code-review/how-to-improve-code-quality/

Pinto de los Ríos, J. S. (Junio de 2015). Implementación del método Kanban en las empresas constructoras pequeñas y medianas en la ejecución de un proyecto en Colombia. 205. Valencia, España.

Pressman, R. (2010). *Ingeniería del Software: un enfoque práctico.* México, D.F.: McGraw-Hill Interamericana.

Robert. (18 de Abril de 2016). *Informática++.* Obtenido de Universidad Oberta de Catalunya: http://informatica.blogs.uoc.edu/2016/04/18/por-que-es-tan-dificil-conseguir-software-de-calidad/

SCRUMstudy. (2016). *Cuerpo de Conocimiento de Scrum.* Arizona.

Seymor, J. (10 de Enero de 2014). *Software Testing Platform provides C# and Java code analysis.* Obtenido de Industry News: http://news.thomasnet.com/

Sharma, T., Mishra, P., & Tiwari, R. (2016). Designite - A Software Design Quality Assessment Tool. *IEEE Xplore*.

Smartsheet. (20 de 05 de 2016). *Smartsheet*. Obtenido de https://www.smartsheet.com/agile-vs-scrum-vs-waterfall-vs-kanban

SonarSource S.A. (2017). *SonarQube*. Obtenido de https://www.sonarqube.org/features/clean-code/

Tanaka, T., Ogasawa, H., Yamada, A., & Aizawa, M. (2002). Software quality analysis and measurement service activity in the company. *IEEE Xplore*.

Universidad La Salle. (2015). Ciencia y Tecnología. *Revista de Ciencia de la Computación*, pp. 23-24.

Wailgum, T. (2010). Siete pasos importantes para mejorar la calidad del software. *CIO Perú*, 32.

Wheeler, D. (14 de Ene de 2017). *Sourceforge*. Obtenido de http://cloc.sourceforge.net/#Overview

ANEXOS

Anexo n.° 1. Ficha de resultados de indicadores - SonarQube

FICHA DE RESULTADOS DE INDICADORES - SONARQUBE	
Solución Analizada	
Fecha	
Versión	
# Ciclo	
Defectos y defectos potenciales	
Incumplimiento de estándares	
Duplicados	
Falta de pruebas unitarias	
Mala distribución de la complejidad	
Código espagueti	
Insuficientes o demasiados comentarios	

Anexo n.° 2. Ficha de resultados de indicadores – Designite

FICHA DE RESULTADOS DE INDICADORES # 1 - DESIGNITE			
Solución Analizada			
Fecha			
Versión			
# Ciclo			
Total # Lines of Code		Total # Namespaces	
Total # Clases		Total # Methods	
Total # Metric Violations		Total Smell Density	
Total Code Duplication			

FICHA DE RESULTADOS DE INDICADORES # 2 - DESIGNITE	
Solución Analizada	
Fecha	
Versión	
# Ciclo	
LOC	
Muestras de olores	
Encapsulación de olores	
Modulación de olores	
Jerarquía de olores	
Densidad de olores	

Anexo n.° 3. Ficha de resultados de indicadores - Cloc

FICHA DE RESULTADOS DE INDICADORES - CLOC	
Solución Analizada	
Fecha	
Versión	
# Ciclo	
Lenguaje	
Archivos	
En blanco	
Comentarios	
Código	

Anexo n.° 4. Ficha de resultados de indicadores – Visual Studio Code Metrics

FICHA DE RESULTADOS DE INDICADORES - VISUAL STUDIO CODE METRICS	
Solución Analizada	
Fecha	
Versión	
# Ciclo	
Índice de mantenibilidad	
Complejidad ciclomática	
Profundidad de herencia	
Acoplamiento de clases	
Líneas de código	

APÉNDICES

Apéndice A: Instalación de SonarQube

1. Descargar el comprimido de la última versión de SonarQube desde la página oficial.
   a. https://www.sonarqube.org/downloads/
2. Descomprimir el archivo en una ruta de fácil acceso, por ejemplo: C:\sonarqube.
3. Crear base de datos en SQL Server, ejecutando el siguiente script:

```
CREATE DATABASE sonar
COLLATE Modern_Spanish_CS_AS
GO
```

4. Editar el archivo **sonar.properties** ubicado en la carpeta **conf**, para configurar el acceso a la base de datos. El archivo debe tener las siguientes líneas descomentadas:

```
sonar.jdbc.username=sonarqube
sonar.jdbc.password=p@ssw0rd
sonar.jdbc.url=jdbc:sqlserver://localhost;databaseName=sonar

sonar.web.host=localhost
sonar.web.context=/sonar
sonar.web.port=9000
```

5. Para instalar SonarQube como servicio Windows, ubicarse en la carpeta **bin\windows-x86-64** y ejecutar los scripts:
   a. InstallNTService.bat
   b. StartNTService.bat
6. Finalmente iniciar sesión en http://localhost:9000/sonar con las credenciales de Administrador de Sistema (admin/admin).

Apéndice B: Instalación de SonarQube Scanner for MSBuild

1. Verificar que .NET Framework v4.5.2+ está instalada.
2. Verificar que Java Runtime Environment 8 está instalado.
3. Descargar el comprimido de la última versión de SonarQube Scanner para MSBuild desde la página oficial.
    a. https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+fo r+MSBuild
4. Descomprimir el archivo en una ruta de fácil acceso, por ejemplo: C:\sonar-scanner-msbuild.
5. Editar el archivo **SonarQube.Analysis.xml**, para configurar la comunicación con el servidor de SonarQube. El archivo debe tener las siguientes líneas descomentadas:

```
<Property Name="sonar.host.url">http://localhost:9000/sonar</Property>
<Property Name="sonar.login">admin</Property>
<Property Name="sonar.password">admin</Property>
```

Apéndice C: Instalación de Designite

1. Verificar que .NET Framework v4.7 está instalada.
2. Descargar el instalador de la página oficial de Designite.
   a. http://www.designite-tools.com/
3. Ejecutar el instalador y elegir ruta de instalación.
4. Esperar hasta finalizada la instalación.

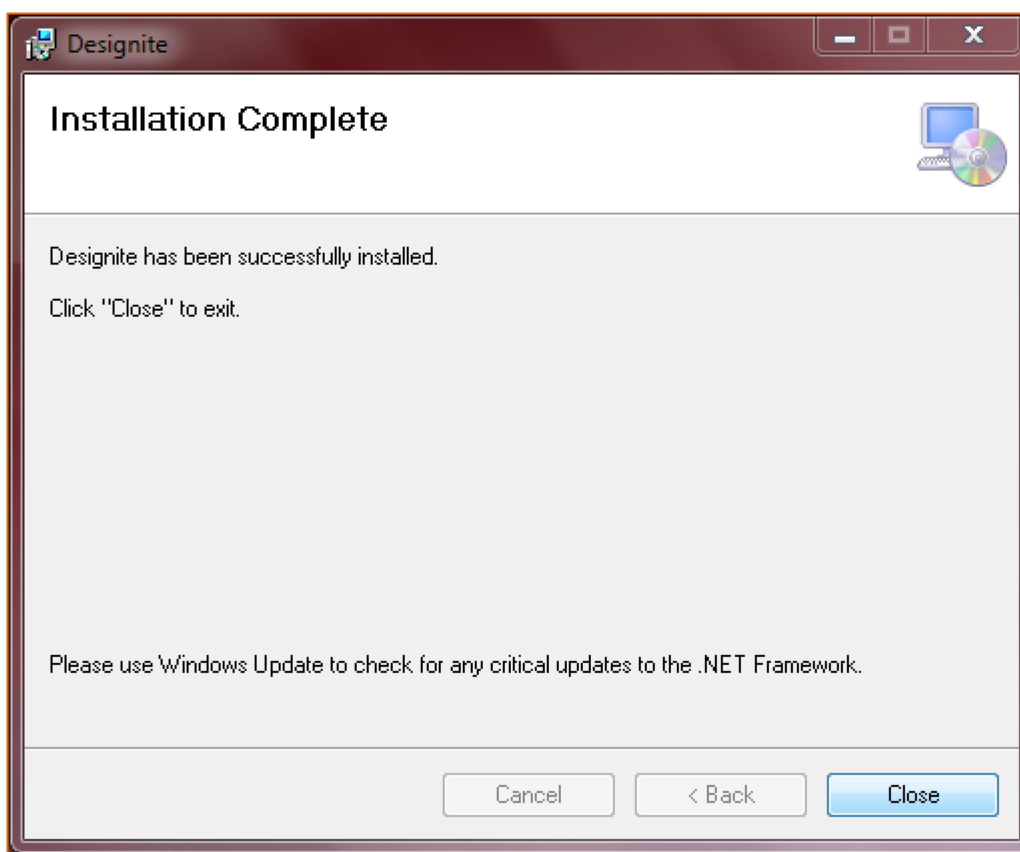


Figura n.° 30. Instalación de Designite

Fuente: Elaboración propia