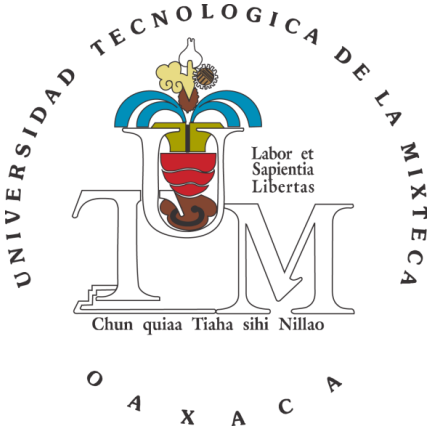


UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

VERIFICACIÓN AUTOMÁTICA DE MÉTRICAS PARA CÓDIGO EN PROYECTOS DE SOFTWARE DISEÑADOS CON EL PARADIGMA ORIENTADOS A OBJETOS

TESIS

PARA OBTENER EL GRADO DE MAESTRO EN COMPUTACIÓN

PRESENTA

ING. RICARDO ROMERO VILLAGÓMEZ

DIRECTOR DE TESIS

DR. CARLOS ALBERTO FERNÁNDEZ Y FERNÁNDEZ

Huajuapan de León, Oaxaca. Enero, 2017

RESUMEN

La calidad del software está estrechamente vinculada con la medición del mismo. La medición de la calidad del software es una necesidad para las empresas de desarrollo de software ya que representa una ventaja estratégica al proporcionar el conocimiento de los procesos productivos y permitir mejoras en las tareas menos eficientes, ya que siempre habrá elementos cualitativos para la creación de software.

Las métricas de software tienen un papel decisivo en la obtención de un producto de alta calidad, porque determinan mediante estadísticas basadas en la experiencia, el avance del software y el cumplimiento de parámetros requeridos.

En el presente trabajo se presenta una propuesta de solución orientada a contribuir a la calidad de proyectos de software diseñados con el paradigma orientado a objetos, automatizando el proceso de verificación de métricas de código, con el objetivo de evaluar, de forma cuantitativa, si ciertas propiedades deseables del diseño orientado a objetos son cumplidas.

CONTENIDO

LISTA DE TABLAS

LISTA DE ILUSTRACIONES

LISTA DE GRAFICAS

INTRODUCCIÓN

La importancia del uso de la Ingeniería de Software (IS) se debe a su enfoque sistemático, disciplinado y cuantificable, aplicado al desarrollo, operación y mantenimiento de software.

Pressman en su libro (Pressman 2010) afirma que "la IS es una tecnología con varias capas, que permite el desarrollo racional y oportuno del software". De acuerdo con Pressman las capas de la IS son: herramientas, métodos, procesos y compromiso con la calidad. La IS se fundamenta en la capa proceso, donde un proceso forma la base para el control de la administración de proyectos, establece el contexto en el que se aplican los modelos técnicos, generando productos de trabajo, define puntos de referencia, se asegura la calidad y se administra el cambio de manera apropiada. La IS es una disciplina que tiene como objetivo la producción de software libre de errores, que se entrega en tiempo y dentro del presupuesto planeado y que satisface las necesidades del cliente, además de que debe de ser fácil de modificar cuando las necesidades del cliente cambien (Schach 2005).

Tradicionalmente el proceso de desarrollo está asociado a un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo el modelado y la documentación detallada. Este esquema tradicional para el desarrollo de software ha demostrado ser efectivo y necesario en proyectos de gran tamaño (respecto a tiempo y recursos). Sin embargo, este esquema no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy variable (requerimientos poco definidos o cambiantes) y en donde se exige reducir drásticamente el tiempo de desarrollo pero manteniendo una alta calidad.

En este contexto la Integración Continua (IC) es uno de los beneficios del Proceso Unificado, sobre todo en las fases de construcción y transición en donde las tareas realizadas en estas etapas son compatibles con las técnicas de IC y del análisis estático de código (Holck & Jørgensen 2007), (Duvall et al. 2007), mientras que para las metodologías ágiles es una de las actividades principales. Para las metodologías ágiles la implementación de la IC es una

de las primeras prioridades en cualquier equipo de desarrollo de software (Crispin & Gregory 2009). En el término IC, la palabra integración se refiere al conjunto de componentes de software y la palabra continuo se refiere a la ausencia de limitaciones de tiempo. La IC puede complementar un enfoque por etapas, donde continua se refiere a la forma en que la integración se lleva a cabo durante la fase de integración y pruebas.

La IC es una práctica donde los desarrolladores de software trabajan en pequeños incrementos, contribuyen con código al proyecto con frecuencia y aseguran que el proyecto se compila y pasa sus pruebas en cualquier momento. La IC es la práctica de hacer pequeños cambios bien definidos a la base de código de un proyecto y obtener retroalimentación inmediata para ver si los bancos de pruebas siguen pasando exitosamente (Deshpande & Riehle 2008), también puede ser usada como un medio de coordinación, en sustitución de documentación detallada de diseño (Holck & Jørgensen 2004), (Humble & Farley 2011), (Roberts 2004), (Rogers 2004).

En este sentido, (Holck & Jørgensen 2004) exploran el uso de la IC en dos proyectos de código abierto con el objetivo de identificar las propiedades esenciales de este enfoque, presumiblemente también relevantes en el contexto del desarrollo comercial. Ambos autores sostienen que además de ayudar a la coordinación, un beneficio importante puede ser la motivación de los desarrolladores (ser capaz de producir resultados rápidos y visibles) y mejora la visión de los desarrolladores en muchas partes del software por la delegación de la responsabilidad de la integración.

La práctica de la IC tiene la capacidad de transformar el desarrollo de software a partir de un conjunto de procesos manuales a una serie lógica de procesos automatizados, reproducibles. El corazón de la IC es el servidor de IC, que es la herramienta y la tecnología que fundamentalmente inicia, realiza, coordina e informa sobre las fases de IC (Crispin & Gregory 2009), (Duvall et al. 2007), (Ritchie 2011).

IC forma parte de la Administración de la Configuración de Software (ACS) – también llamado *Software Configuration Management* (*SCM*) – la cual ayuda a mantener la integridad de los componentes de software en todo el ciclo de desarrollo, esto significa asegurar la consistencia de todos los productos (sistema, fuentes, documentación, manuales) con la

información que contienen. Al existir una modificación en algún componente, este cambio se verá reflejado en todos y cada uno de los documentos implicados, por lo que es una de las actividades más importantes en el desarrollo de software (Duvall et al. 2007), (Humble & Farley 2011), (Pressman 2010), (Rehn 2015), (Roberts 2004).

La Organización Internacional de Estandarización (ISO), en su norma ISO 9000:2005 define la calidad como el grado en que un conjunto de características inherentes cumple con los requerimientos (ISO 2015). Para mantener el nivel requerido de la calidad, la seguridad de que los atributos de calidad de software se cumplen es un aspecto importante de la ejecución de proyectos de software. Los atributos de calidad a menudo se evalúan a través de métricas de software.

Según Herbold, Grabowski y Waack (Herbold et al. 2011), afirman que todo software generado, producto de la aplicación de la IS, debe de tener cierto grado de calidad, donde la medición es el proceso por el que los números o símbolos son asignados a los atributos de las entidades del mundo real, de una manera tal, que se describen de acuerdo a reglas claramente definidas. Una manera de medir el software es utilizando métricas.

Las métricas de software son definidas por el IEEE en la norma IEEE 1990 como la medida cuantitativa del grado en que un sistema, componente o proceso poseen un atributo dado. Esto significa que una métrica de software es una regla claramente definida, que asigna valores a entidades de software (por ejemplo, componentes, clases o métodos) o atributos de los procesos de desarrollo. A menudo, una métrica es insuficiente para analizar de forma eficaz un atributo de calidad, por lo que, en ocasiones, se utiliza un conjunto de métricas para determinar si un atributo de calidad se ha cumplido o no.

Por otro lado, Singh y Kahlon (Singh & Kahlon 2014) sostienen que las métricas pueden ser aplicadas en el mantenimiento, pruebas y durante la evolución del software para una variedad de propósitos. Diversos estudios de investigación han diseñado modelos de métricas para el análisis de la calidad del software, sin embargo, es difícil evaluar la calidad de software con un único valor de la métrica, por lo que toma sentido, el uso de umbrales en las métricas.

Para los autores Abu Asad e Izzat (Abu Asad & Izzat 2014), las métricas de software se utilizan como indicadores de la calidad del software desarrollado. Dichas métricas se pueden obtener desde cualquier parte de software, tales como: el código, el diseño o los requerimientos. Estos autores han propuesto y evaluado métricas para el diseño y el código de software, en donde utilizan los informes de errores para su posterior análisis y asociación, además de usar datos estadísticos usando métodos de minería de datos para recolectar y analizar de forma automática las métricas.

Para los autores Wang, Khoshgoftaar y Liang (Wang et al. 2013), sostienen que las métricas de software (características o atributos) se recogen durante el ciclo de desarrollo de software. La selección de métricas es uno de los pasos de pre procesamiento más importantes en el proceso de construcción de modelos de predicción de defectos y puede mejorar el resultado final.

Hasta ahora, se han propuesto muchas métricas de software orientado a objetos y se aplica a los proyectos de software. Es bien sabido que dos de los conjuntos de métricas ampliamente aceptados son las métricas de Chidamber y Kemerer (CK) y las métricas para sistemas orientados a objetos (MOOD) (Cheikhi et al. 2014), (Greiner et al. 2010), (Ma et al. 2010), (Wang et al. 2013).

En su libro Gorton (Gorton 2011) sostiene que la base para generar un software de calidad, reside en su arquitectura, donde la arquitectura de software va más allá de los algoritmos y las estructuras de datos, se enfoca en el diseño y la especificación de la estructura general de un sistema, ésta última incluye la organización y estructura del control global del sistema, los protocolos de comunicación, la sincronización y acceso a datos, la asignación de funcionalidad a elementos de diseño, la distribución física de los componentes del sistema, el escalamiento y el desempeño y a la selección de alternativas de diseño. La arquitectura debe de cumplir con atributos de calidad.

Los atributos de calidad definen los requerimientos de un sistema en términos de escalabilidad, disponibilidad, facilidad de cambio, la portabilidad, facilidad de uso, rendimiento, seguridad, entre otras, además de ser parte de los requerimientos no funcionales

de un sistema, ya que todo sistema tendrá requerimientos no funcionales que pueden ser expresados en términos de atributos de calidad.

En este trabajo se pretende ampliar la funcionalidad del servidor de IC para obtener métricas de Chidamber y Kemerer, así como las métricas para el diseño orientado a objetos, ya que dichas métricas han demostrado ser buenos predictores de la complejidad del diseño orientado a objetos, así como de los defectos de software (Herbold et al. 2011); las métricas de software orientado a objetos juegan un papel importante para garantizar la calidad deseada de los sistemas de software.

En seguida se da una breve descripción del contenido de este documento. En la sección 2 se presenta el marco teórico conceptual que será usado para cumplir con el objetivo general de este documento; se exponen las definiciones de conceptos como IC, control de versiones, herramientas de análisis estático, servidores de IC, la metodología orientada a objetos en donde se detallan los factores de calidad, así como las características de dicho paradigma, y finalmente, las métricas de calidad de software detallando las fórmulas para obtener las métricas CK y MOOD. En la siguiente sección se presenta el análisis de las herramientas y la definición de la metodología a emplear en esta investigación; se expone la evaluación de algunas herramientas de análisis estático que existen en el mercado, se presenta el análisis de la arquitectura del servidor de IC llamado *Jenkins* y se presenta el análisis realizado a la herramienta *SonarQube* que es una herramienta especializada en la gestión de la calidad de software; finalmente se presenta la metodología *OpenUP*, la cual será usada para el desarrollo de esta investigación. En la sección 4 se presenta el desarrollo de la investigación en base a la metodología empleada, cubriendo las fases de inicio, elaboración, construcción y transición. En la sección 5 se presenta el resultado de las métricas obtenidas por el componente creado, se seleccionaros 5 proyectos java, que superan las 50,000 líneas de código. En la siguiente sección se presentan las conclusiones de la tesis y el trabajo futuro. En la sección de apéndices se presentan los artefactos definidos por la metodología *OpenUP*. En el apéndice 1 se encuentran los artefactos referentes a la fase de inicio; en el apéndice 2 se presentan los artefactos de la fase de elaboración; en el apéndice 3 tenemos los entregables de la fase de construcción y finalmente, el apéndice 4 está conformado por los entregables de la fase de transición. En la última sección se describen los libros, documentos y artículos

que fueron citados en el presente documento, así como la bibliografía fundamental de ésta investigación.

1.- PLANTEAMIENTO DEL PROBLEMA Y METODOLOGÍA

1.1.- Problemática

En los últimos años hemos visto un incremento considerable del uso del paradigma orientado a objetos en el desarrollo de software. El uso de técnicas de desarrollo de software orientado a objetos introduce nuevos elementos a la complejidad del software, tanto en el proceso de desarrollo de software como en el producto final.

La columna vertebral de cualquier sistema de software es su diseño y el paradigma orientado a objetos incluye un grupo de mecanismos como la herencia, encapsulación, polimorfismo y el paso de mensajes que se crean para permitir la construcción de diseños en los que se hacen cumplir esas funciones.

Muchas métricas orientadas a objetos se han propuesto específicamente con el fin de evaluar el diseño de un sistema de software. Sin embargo, la mayoría de los enfoques existentes para la medición de estos parámetros de diseño implican sólo algunos de los aspectos del paradigma orientado a objetos.

Las métricas aquí planteadas (CK y MOOD) tratan de captar los diferentes aspectos del software como producto y su proceso de desarrollo. Estas métricas se pueden aplicar en proyectos creados mediante el lenguaje *Java* para evaluar y comparar el diseño del código.

En la actualidad existe una gran cantidad de aplicaciones de análisis estático (el análisis estático examina el código fuente sin ejecutarlo) aplicado a diversas tecnologías, entre las cuales se pueden citar las siguientes:

- *Kiuwan Software Analytics*
- *HP Fortify Source Code Analyzer*
- *IBM Rational AppScan Source Edition*
- *Team Foundation Server*
- *CAST Application Intelligence Platform*

Características de Herramientas en el Mercado

Kiuwan Software Analytics

Kiuwan Software Analytics mide varias métricas de código intrínsecas en cada análisis. Se encuentran organizadas en varias categorías en función de su naturaleza:

- Tamaño. Eso incluye el tamaño físico y métricas de tamaño funcional.
- Métricas relacionadas con la complejidad. Incluyendo la complejidad ciclomática y *fan-out*.
- Documentación. Medición del código documentado.
- Calidad. Cumplimiento de la normativa de las mejores prácticas definidas en las métricas del modelo de software, número de defectos o vulnerabilidades, reglas de violaciones.
- Gobernabilidad. En cuanto a la exposición al riesgo de sus esfuerzos de desarrollo.

Se puede acceder a toda la información sobre las métricas del código, incluyendo la evolución de los indicadores más relevantes, para evaluar la mejora en los equipos de desarrollo.

También genera un reporte de métricas (en formato XLS), que contiene todas las métricas calculadas, tanto a nivel proyecto como a nivel de archivo (Kiuwan 2015).

HP Fortify Source Code Analyzer

HP Fortify Source Code Analyzer es toda una suite de análisis de código estático automatizado para ayudar a los desarrolladores a eliminar las vulnerabilidades y crear software seguro, el módulo *HP Fortify Static Code Analyzer* (*SCA*), ayuda a comprobar que su software sea de confianza, a reducir costos, a aumentar la productividad y a implementar las prácticas recomendadas de codificación segura. El módulo de pruebas de seguridad de aplicaciones estáticas (*SAST*), identifica las causas raíz de las vulnerabilidades de la seguridad del software, correlaciona y prioriza los resultados. Así le facilita una guía para cerrar brechas en la seguridad. Para comprobar que los problemas más serios se han

solucionado primero, relaciona y prioriza los resultados para ofrecer una lista de problemas por orden y en forma precisa (HP 2015).

IBM Rational AppScan Source Edition

IBM Rational AppScan Source Edition es una suite de herramientas que ofrecen el mejor valor a cada usuario que esté relacionado con seguridad de software. Tanto para analistas de control de calidad o seguridad, desarrollador o ejecutivo, los productos de la suite ofrecen la funcionalidad, la flexibilidad y la potencia adecuada que necesita para su escritorio.

El conjunto de productos de la suite incluye:

Rational AppScan Source Edition for Security: Es un entorno de trabajo para analizar, aislar y actuar en las vulnerabilidades prioritarias.

Rational AppScan Source Edition for Automation: Le permite automatizar aspectos clave del flujo de trabajo y de las exploraciones integradas de *Rational AppScan Source Edition* con entornos de compilación durante el ciclo de vida de desarrollo del software.

Rational AppScan Source Edition for Developer: Los conectores del desarrollador integran muchas características de *Rational AppScan Source Edition for Security* en *Visual Studio*, en *Eclipse* y en *Rational Application Developer*, lo que permite a los desarrolladores de software encontrar y tomar medidas sobre las vulnerabilidades durante el proceso de desarrollo. El conector de Eclipse le permite explorar código fuente en busca de vulnerabilidades y riesgos de seguridad, crear archivo de configuración de reglas de seguridad que habilitan la exploración de seguridad en la interfaz de línea de mandatos de *Rational AppScan Source Edition* y *Rational AppScan Source Edition for Automation* (IBM 2015).

En relación a las métricas de calidad, incluye tres configuraciones de reglas de calidad:

Revisión de código básica de C/C++ (*codereview_basic_cpp.quality*): Esta configuración de reglas de calidad es la misma que las reglas de calidad críticas de C/C++ (*critical_rules_cpp.quality*).

Revisión de código básica de *Java* (*codereview_basic_java.quality*): Esta configuración incluye las reglas de revisión de código java recomendadas (las cuales fueron diseñadas para generar únicamente resultados de máxima gravedad).

Revisión de código ampliada de *Java* (*codereview_extended_java.quality*): Esta configuración de reglas incluye las mismas reglas que las que se encuentran en la revisión de código básica, más reglas para encontrar problemas de baja seguridad y de análisis de flujo de datos de *Java*.

Team Foundation Server (TFS)

TFS es una suite de *Microsoft* que permite gestionar todo el ciclo de vida de tu desarrollo, incluyendo métricas sobre el histórico del proyecto. Al crear un nuevo proyecto, *TFS* genera un conjunto de reportes estándar de acuerdo con las especificaciones de la plantilla de proceso. Los reportes elaborados por *TFS* permiten evaluar rápidamente el estado del proyecto de equipo, la calidad del software en fase de desarrollo y el progreso del proyecto. Estos reportes resumen las métricas de las cosas tales como elementos de trabajo, control de código fuente, resultados de pruebas, y compilaciones.

La métricas de calidad se obtiene al ejecutar *PowerTools CodeMetric*, esta herramienta permite analizar conjuntos de código con el fin de entender las métricas de complejidad asociados con el ensamblaje del código de una aplicación; encapsula el código necesario para hacer el análisis real de la aplicación; publica los resultados de las métricas en *TFS* (Microsoft 2015).

CAST Application Intelligence Platform (AIP)

AIP es una solución de análisis y medición de la calidad del software de nivel empresarial diseñado para analizar aplicaciones en diferentes tecnologías y de múltiples niveles para las vulnerabilidades técnicas y la adhesión a los estándares arquitectónicos y de codificación. Esta herramienta proporciona una vista de abajo arriba de la deuda técnica, documenta automáticamente los sistemas complejos y/o heredados y proporciona información en tiempo real para mejorar el rendimiento de la salud de aplicaciones y equipo de desarrollo.

Como una solución totalmente automatizada, *AIP* permite la evaluación continua y objetiva de la calidad estructural de las aplicaciones de software. A nivel de componentes y aplicaciones, calidad estructural de la información procesable a todos los niveles de una organización de tecnologías de la información. Esto permite una gestión proactiva del riesgo, mejora de la productividad y reducción de costos; realiza el análisis en varias tecnologías, entre las cuales puedo citar *J2EE, 4GL, SAP*, *.NET*, *Cobol*.

La robustez es un indicador de la probabilidad de que en una aplicación incurrirá defectos, datos corruptos o fallar por completo en un ambiente productivo, *AIP* mide la robustez basándose en las mejores prácticas de la industria, verifica la complejidad algorítmica y de control de flujo, el acceso a datos controlados a nivel de arquitectura, diseño orientado a objetivos arquitectónicos, el nivel de acoplamiento y las dependencias (CastSoftware 2015).

Todas las aplicaciones antes descritas son sistemas propietarios (con costo para el cliente) y puede ser distribuido de forma local o como *Software as a Service* (*SaaS*), el cual es un modelo de distribución de software, donde el soporte lógico reside en servidores de la empresa proveedora y el cliente ingresa a la aplicación vía internet. Realizan el análisis de sistemas de software, la medición de la calidad y la seguridad de las mismas. Ofrecen todas las métricas relevantes para cada aplicación (costo, esfuerzo, calidad, facilidad de mantenimiento, eficiencia y dependencia del software), que se puede usar como referencia para la toma de decisiones en una empresa de desarrollo de software, permitiendo la reducción de costos, la mitigación de riesgos, medición objetiva, certificación técnica de la calidad del software, por mencionar algunas.

En la actualidad no existe una herramienta de la categoría software libre que obtenga las métricas CK y MOOD, por lo que este trabajo propone un componente de software, construido en lenguaje *Java*, que permita realizar el análisis del código desde el punto de vista estático, es decir, analizar el código solamente revisando los archivos fuente y que se integre a servidores de IC.

1.2.- Planteamiento del problema

En los últimos años hemos asistido a un crecimiento exponencial de la demanda de software, que se ha venido aplicando en la resolución de tareas cada vez más complejas y proporcionando cada vez mayor valor añadido (Pressman 2010). Los productos de software siguen entregándose fuera de tiempo, se exceden en costo y no cumplen con la calidad esperada por el cliente (Ritchie 2011). Debido a estas circunstancias, en los últimos años se están desarrollando una serie de modelos y metodologías (entre ellas *Scrum*, *XP*) orientadas a minimizar la problemática a la que enfrenta la gestión de proyectos, las cuales hacen énfasis en las ventajas de la IC que incluyen la reducción de los riesgos de integración, la detección temprana de errores (ya que cuando todos los módulos de un sistema se combinan, por lo general resulta en un gran número de errores, difícil de aislar y corregir debido al tamaño del sistema), la automatización de pruebas (unitarias, funcionales, de aceptación, de integración, entre otras), la coordinación del equipo de desarrollo.

El presente trabajo propone extraer las métricas CK y MOOD de proyectos creados en lenguaje *Java* y que usen el paradigma orientado a objetos, para ser usados como referencia para la toma de decisiones, para la reducción de costos y para la mitigación de riesgos en empresas de desarrollo de software. Para lograr esto, los proyectos deben recoger gran parte de los datos en sí y en un nivel lo suficientemente bajo para que los datos puedan ser claramente entendidos y se puedan representar en muchas maneras diferentes, por ejemplo: para contar líneas de código; para obtener el número de paquetes, clases y métodos; para obtener información sobre los defectos encontrados.

1.3.- Hipótesis, objetivos y metodología

1.3.1.- Hipótesis

A partir de lo anteriormente expuesto, la hipótesis planteada es:

Es posible, en el marco de trabajo de los servidores de IC, desarrollar nueva funcionalidad que permita recolectar información de proyectos de software orientados a objetos, para obtener las métricas CK y MOOD de forma automatizada.

1.3.2.- Objetivos

El objetivo general del presente trabajo es el de desarrollar un componente para servidores de IC, que mediante herramientas de análisis estático, recolecte información de proyectos *Java* de software que apliquen el paradigma de la programación orientada a objetos, para generar las métricas CK y MOOD, estas métricas exploran la calidad del código de componentes de software que han sido diseñados en el paradigma de programación orientado a objetos.

Los objetivos específicos son:

Investigar la arquitectura del servidor de IC más usado en la actualidad.

Evaluar las herramientas de análisis estáticos existentes en la actualidad y crear un cuadro comparativo que permita identificar las características más usadas y útiles con el fin de seleccionar la herramienta de análisis estático a usar.

Diseñar y construir un componente que integre las herramientas de análisis estático para recolectar la información, para generar métricas CK y MOOD, y realizar el análisis correspondiente de ellas.

Instalar en al menos dos nodos de red el componente, para ser configurado y ejecutado desde el servidor de IC.

Presentar un reporte de las métricas obtenidas por cada proyecto de la aplicación. La información mostrada en el reporte será además de las métricas normales (número de líneas de código, número de paquetes, número de clases, número de métodos, entre otras), las métricas de CK (*WMC, DIT, NOC, CBO, RFC, LCOM, LCOM1, LCOM2, MPC, DAC) y MOOD (PF, CRF, MHF, AHF, MIF, AIF*).

Proporcionar diferentes formatos para exportar el análisis realizado (*CSV, PDF, XLS*).

1.3.3.- Metodología

Para desarrollar esta investigación se realizará como primer paso una búsqueda de literatura en bases de datos como *EBSCO, SPRINGER, SCIENCEDIRECT, DIRECTORY OF OPEN ACCESS JOURNALS*. Las palabras claves para las búsquedas serán: "*Continuous Delivery*", "*Software Architecture*", "*Software Architecture Analysis*", "*Architecture and Patterns*", "*Quality Metrics*", "*Software Quality Metrics*", "*Software Metrics*", "*CK Metrics*", "*CK MOOD*". Como segundo paso se realizará una búsqueda y análisis documental de los servidores de IC más populares, para conocer su arquitectura y la forma de implementar el componente a desarrollar. En el tercer paso se realizará el diseño y la implementación del componente de software, que recolectará información para la verificación de las métricas CK y MOOD aplicados al código de sistemas de software que usen en su diseño el paradigma orientado a objetos. Como cuarto paso, la actividad principal se basará en la selección de los proyectos en donde será aplicado el componente, el número de proyectos seleccionado será de por lo menos 20 proyectos de software que hayan sido desarrollados en el lenguaje de programación *Java* usando *Maven* y que hayan sido diseñados usado el paradigma orientado a objetos. En el quinto paso se evaluará el componente desarrollado en los proyectos seleccionados en el punto anterior en por lo menos 5 ocasiones, para comprobar la generación de datos históricos. En el penúltimo paso se analizarán y documentarán los resultados obtenidos y se comprobará la hipótesis de este trabajo. Como último paso se generarán las conclusiones en base al análisis de los resultados alcanzados.

2.- MARCO TEÓRICO CONCEPTUAL

2.1.- Integración Continua

El concepto de IC –*Continuous Integration*– tomaron relevancia a partir de un conocido artículo de Martin Fowler. El éxito de este sistema puede comprobarse, en el gran número de herramientas existentes, tanto de código abierto como comerciales, que llevan a cabo esta tarea. Hay que destacar que el proceso de IC supone algo más que utilizar una herramienta ya que conlleva adoptar nuevas buenas prácticas. Esta práctica se asocia a las metodologías ágiles como *Extreme Programming* (Crispin & Gregory 2009), (Fowler 2015).

Un sistema de IC consiste en un ciclo que básicamente se encarga de revisar los cambios existentes en el repositorio, compilar automáticamente los proyectos y ejecutar pruebas automáticas de forma frecuente. Esto permite detectar antes los fallos, y evitar que tenga que ser el propio usuario el que compile el proyecto cada cierto tiempo. Con esto se elimina el riesgo de que en algunas fases el código no compile, no fueron ejecutadas las pruebas y se haya escrito mucho código cuando un error es detectado. Hay que tener en cuenta que en este proceso no participa activamente el desarrollador, aunque puede forzar a que se produzcan compilaciones aparte de las automáticas. Cuando un sistema de IC detecta un fallo lo comunicará a los interesados con rapidez y esta supervisión será la que mejore el conocimiento de los fallos y en general el proceso de desarrollo (Duvall et al. 2007).

En ocasiones implantar un sistema de IC puede no tener demasiado sentido: si no se han elaborado pruebas automáticas y la fase de compilación es rápida puede prescindirse de esta metodología. Cuanta mayor es la envergadura del proyecto y mayor número de desarrolladores trabajan en él, más conveniente será para el proyecto. Para implantar un sistema de IC se debe planificar la frecuencia con la que deben compilarse y ejecutarse las pruebas de los proyectos. Este tiempo debe ser como mínimo de una vez al día, y se recomienda que sea mucho más a menudo (incluso es habitual que se ejecute cada vez que se detecten cambios en el repositorio). El servidor –*build server*– estará conectado con el repositorio del código –*repository server*– de modo que en cada momento programado revisa si hay nuevos datos en el repositorio, y si es así compila el proyecto. Este proceso en general será visible para todos los usuarios de alguna forma –la más habitual es por medio de una

interfaz web, aunque también incluso en una pantalla externa que pueda ver el equipo de desarrollo–, y desembocará en un resultado –éxito o fracaso de la compilación–. Posteriormente ejecutará las pruebas automáticas, y si se pasan con éxito, esta rama principal es estable y puede seguir trabajándose en ella. En caso contrario, ha ocurrido un error, y puede volverse al código anterior, responsabilizando al desarrollador que incorporó los últimos cambios para que arregle el código en una rama propia y mantener la rama principal disponible para compilación y en pleno funcionamiento. Se le puede avisar por correo electrónico, chat, rss, entre otras opciones, a esa persona o a todas las que estén asignadas a un determinado proyecto, también puede tratar de solucionarse el fallo en la rama principal durante un tiempo, pero la idea que hay debajo es que la rama principal esté casi siempre libre de defectos e incidencias (Rehn 2015).

La IC representa un cambio de paradigma. Sin integración continua, el software se encuentra aislado hasta que alguien demuestre que funciona, por lo general eso sucede durante una etapa de pruebas o la etapa de integración. Con la integración continua, el software está demostrando que funciona (suponiendo un conjunto suficientemente amplio de pruebas automatizadas) con cada nuevo cambio subido al sistema de control de versiones. Los equipos que utilizan la IC efectivamente son capaces de ofrecer un producto de software más rápido y con menos errores, que los equipos que no lo hacen. Los errores son detectados en fases tempranas, proporcionando un ahorro de costos y de tiempo significativos. Por lo tanto se puede considerar que es una práctica esencial para los equipos profesionales, tal vez tan importante como el uso de control de versiones (Humble & Farley 2011).

En la Ilustración 1 los rectángulos sin sombrear representan a subsistemas discretos y a funcionalidad dentro del sistema. Las flechas muestran el flujo de información entre los distintos componentes. La nube representa la ejecución remota de los procesos de construcción. Los rectángulos sombreados representan el acoplamiento entre los subsistemas; por ejemplo, el monitoreo puede incluir la supervisión del proceso de construcción en sí y algunos aspectos del sistema como pueden ser el uso de CPU, el uso de memoria, el uso de I/O.

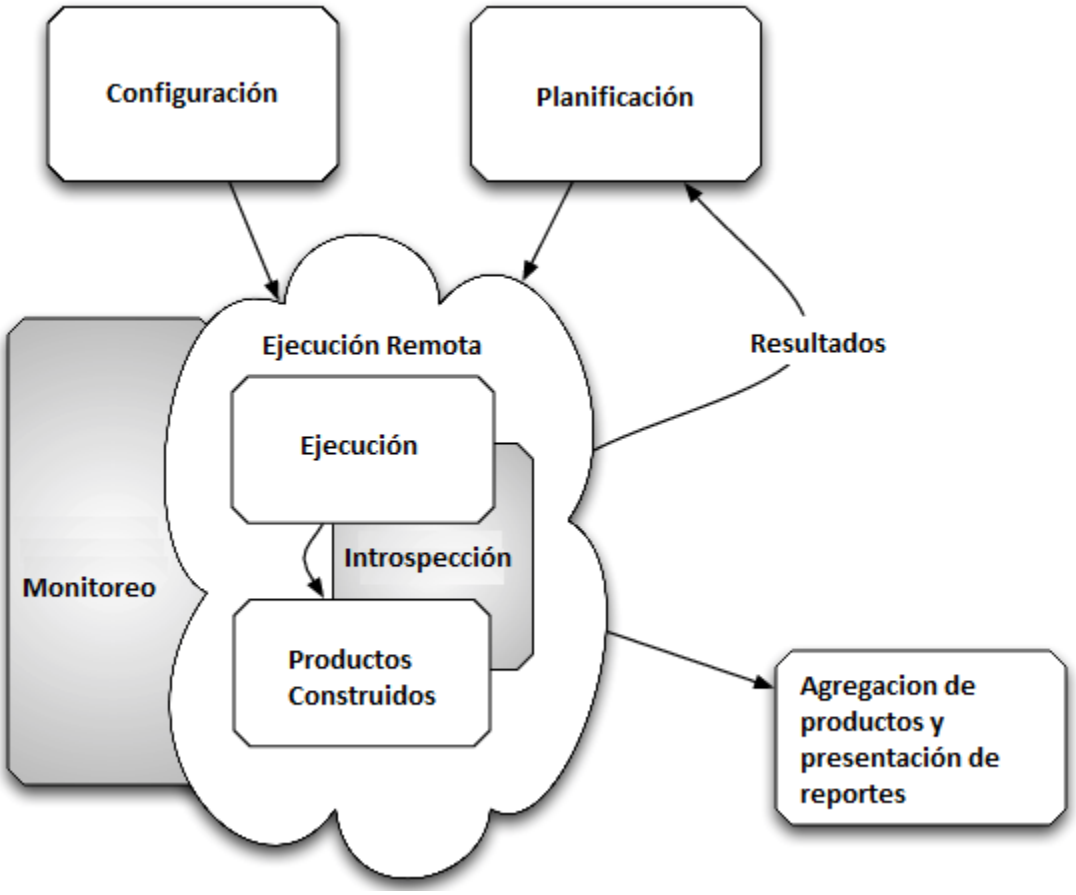


Ilustración 1 Estructura interna de un servidor de IC
Ferguson, (2011)

Las arquitecturas de los servidores de IC se encuentra dominado por dos tipos: las arquitecturas maestro/esclavo, en la que un servidor central dirige y controla a distancia la ejecución de tareas en servidores remotos; y la arquitectura centralizada, en el que un servidor central agrega los reportes generados por las herramientas clientes a un sitio central (Berg 2015).

Implantación de un entorno de IC

El procedimiento para la implantación del proceso de IC propone 3 elementos fundamentales como parte del proceso de IC: el repositorio de código fuente, la construcción automatizada del proyecto y un acuerdo del equipo de desarrollo.

Herramienta de control de versiones

Todo el proyecto debe ser revisado en un único sistema de control de versiones: el código, las pruebas, los scripts de base de datos y todo lo necesario para crear, instalar, ejecutar y probar la aplicación. Algunas personas no consideran que su proyecto lo suficientemente grande como para justificar el uso de control de versiones. No existe un proyecto lo suficientemente pequeño como para prescindir de él. Existen varios sistemas de control de versiones que son sencillos, potentes, ligeros y de código libre.

Herramienta de construcción automatizada

El proyecto debe de iniciar su construcción a partir de dicha herramienta, estas herramientas pueden realizar las tareas de construcción, compilación, ejecución de prueba o puede seleccionar una compleja colección de scripts de construcción de varias etapas que llaman unos a otros. Cualquiera que sea el mecanismo, debe ser posible ejecutar su construcción, pruebas y proceso de implementación de forma automatizada. Los ambientes integrados de desarrollo - *IDE* (*Integrated Development Environment*) por sus siglas en inglés y las herramientas de integración continua se han vuelto muy sofisticadas en estos días y normalmente se puede construir el software y ejecutar pruebas sin usar la línea de comandos. El uso de ésta herramienta se vuelve más importante cuanto más complejo sea el proyecto en el que se esté trabajando; hace que la comprensión, el mantenimiento y la depuración de la construcción sean más fácil y permite una mejor colaboración con el equipo de desarrollo. En resumen, ésta herramienta debe:

Ser capaz de ejecutar el proceso de construcción en una forma automatizada en el entorno de IC para que pueda ser auditado.

Tratar los scripts de construcción como el código base. Estos deben ser probados y constantemente refactorizados para que sean ordenados y fácil de entender.

Acuerdo del equipo de desarrollo

La integración continua es una práctica, no es una herramienta. Se requiere un grado de compromiso y de disciplina del equipo de desarrollo. Es necesario que todo el mundo genere

cambios incrementales con cierta frecuencia al código base. Si las personas no adoptan la disciplina necesaria para que funcione, los intentos de integración continua no darán lugar a la mejora de la calidad que se espera.

2.2.- Control de versiones

Humble y Farley (Humble & Farley 2011) dejan en claro que la gestión de la configuración es un término que es utilizado ampliamente, a menudo como sinónimo de control de versiones, por lo que es conveniente establecer su definición. Gestión de la configuración se refiere al proceso por el cual todos los artefactos pertinentes a un proyecto y las relaciones entre ellos, son almacenados, recuperados, identificados de forma única y con la posibilidad de ser modificados. La estrategia de gestión de la configuración determinará la forma de gestionar todos los cambios que ocurren dentro del proyecto. Por lo tanto, debe registrar la evolución de sus componentes y aplicaciones. También gobernará cómo su equipo colabora en la construcción de la aplicación.

Aunque los sistemas de control de versiones son la herramienta más obvia en términos de la gestión de la configuración, la decisión de utilizar una de éstas herramientas es sólo el primer paso en el desarrollo de una estrategia de gestión de la configuración.

Los sistemas de control de versiones (SCV) permiten a grupos de personas trabajar de forma colaborativa en el desarrollo de proyectos, frecuentemente a través de Internet. Son sistemas que ponen marcas en las diferentes versiones para ser identificadas posteriormente, facilitan el trabajo en paralelo de grupos de usuarios, permiten analizar la evolución de los diferentes módulos del proyecto, y mantienen un control detallado sobre los cambios que se han realizado; funciones que son indispensables durante la vida del proyecto.

Estos sistemas no sólo tienen aplicación en el desarrollo del software, sino que además son ampliamente utilizados en la creación de documentación, sitios web y en general en cualquier proyecto colaborativo que requiera trabajar con equipos de personas de forma concurrente. Los sistemas de control de versiones se basan en mantener todos los archivos del proyecto en un lugar centralizado, normalmente un único servidor, aunque también hay sistemas distribuidos, donde los desarrolladores se conectan y descargan una copia en local del

proyecto (Chacon 2015), (Deshpande & Riehle 2008), (Duvall et al. 2007), (Loeliger 2009), (Rehn 2015).

Al SCV, se envían periódicamente los cambios que realizan al servidor y van actualizando su directorio de trabajo que otros usuarios a su vez han ido modificando. Los sistemas de control de versiones de código están integrados en el proceso de desarrollo de software de muchas empresas. Cualquier empresa que tenga más de un programador trabajando en un mismo proyecto acostumbra a tener un sistema de este tipo, y a medida que crece el número de personas que se involucran en un proyecto, más indispensable se hace un sistema de control de versiones.

Los SCV modernos suelen aportar una amplia gama de funcionalidades, de las cuales se extraen continuación las más relevantes:

Control de usuarios. Establecer los usuarios que tienen acceso a los archivos del proyecto y qué tipo de acceso tienen asignado.

Mantener un control detallado de los cambios realizados en cada archivo. Disponer de un control de los cambios efectuados en el archivo, fecha, causa que los motivó, quién los realizó y mantener las diferentes versiones.

Resolver conflictos de actualización de archivos. En caso de que dos usuarios modifiquen un archivo proporcionar herramientas para gestionar este tipo de situaciones.

Bifurcación de proyectos. A partir de un punto concreto, crear ramas del proyecto para que puedan evolucionar por separado. Es habitual utilizar esta técnica cuando se ha liberado una versión de un producto y se requiere que siga evolucionando.

En resumen, los SCV tienen 2 objetivos en concreto: el primer objetivo es el de conservar y proporcionar acceso a todas las versiones de cada archivo que se haya almacenado en el repositorio. Estos sistemas también proporcionan una forma para que los metadatos (la información que describe los datos) se adjuntarán a archivos individuales o grupos de archivos almacenados. El segundo objetivo es el permitir a los equipos que pueden ser

distribuidos a través del espacio y el tiempo para colaborar simultáneamente (trabajo colaborativo) (Humble & Farley 2011).

En un ambiente de IC, el servidor de IC es el encargado de interactuar y supervisar el sistema de control de versiones para los comprobar si existen cambios en el código base de una aplicación de software. Cada vez que se detecta un cambio, esta herramienta compila dicho código automáticamente y ejecuta las pruebas que hayan sido creadas para la aplicación.

Por la forma en que la información contenida en los proyectos es compartida y manipulada, los SCV se clasifican en centralizados y distribuidos (Chacon 2015), (Miravet et al. 2011), (Loeliger 2009).

En la Ilustración 2 se muestran los SCV centralizados, los cuales se caracterizan por contar con un servidor central de donde los desarrolladores toman información de alguna versión del proyecto, la manipulan y al finalizar el proceso de desarrollo, la actualizan en el servidor central.

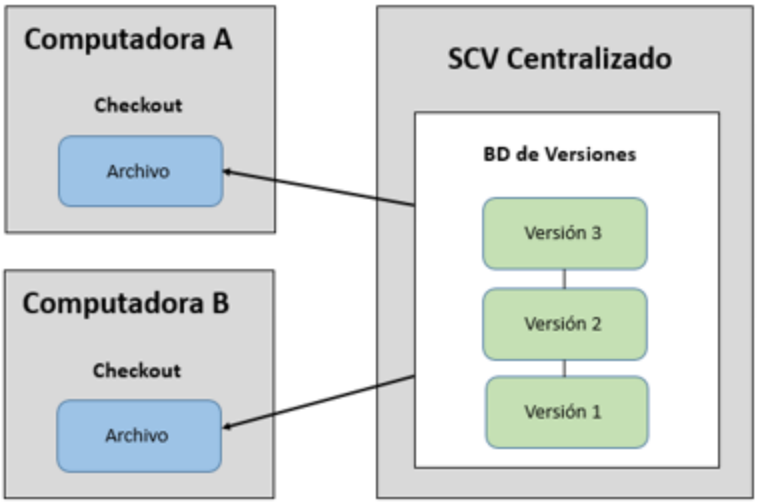


Ilustración 2 SCV Centralizado
Chacon, (2015)

Por otro lado, en la Ilustración 3 se presenta el esquema de trabajo de los SCV distribuidos. Estos sistemas no necesitan un servidor central para almacenar la información, sino que pueden disponer de alguna versión y trabajar localmente con la información, generando nuevas versiones, sin necesidad de almacenar la versión resultante en un servidor central.

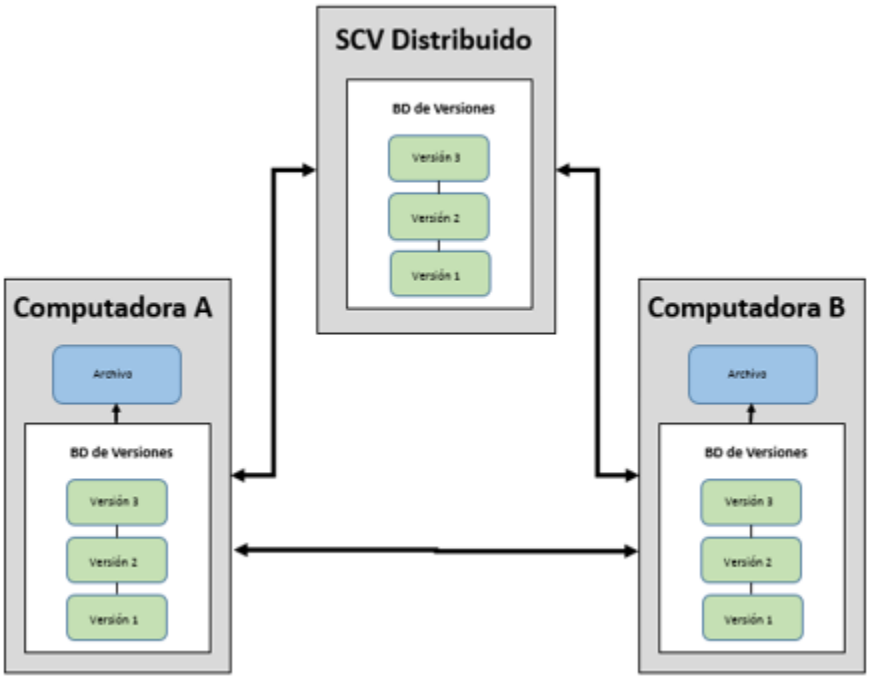


Ilustración 3 SCV Distribuido
Chacon, (2015)

En base al artículo publicado en *Smashing Magazine* (Stansberry 2008) son presentados los mejores sistemas de control de versiones de código abierto y de algunas herramientas que facilitan la creación de un sistema de control de versiones. Entre los mejores sistemas de control de versiones podemos mencionar a *CVS*[1], *SVN*[2], *Git*[3], *Bazaar*[4], *Monotone*[5].

[1] CVS página web http://www.nongnu.org/cvs/
[2] SVN página web http://subversion.tigris.org/
[3] Git página web http://git-scm.com/
[4] Bazaar página web http://bazaar.canonical.com/en/
[5] Monotone página web http://www.monotone.ca/

2.3.- Herramientas de análisis estático

La calidad del software depende principalmente de los defectos que se introduzcan en la fase de codificación. Por lo tanto cualquier técnica que permita eliminar estos defectos en la fase de creación permitirá aumentar la calidad a un costo reducido. Las técnicas de Análisis Estático realizan esa función permitiendo localizar defectos sin ejecutar el código. Existen diversas técnicas y no siempre se pueden aplicar todas por razones de costo y tiempo.

La técnica del Análisis Estático de Código aparece recomendada en la mayoría de las normas de desarrollo de sistemas, su uso es altamente recomendable no sólo en el desarrollo de sistemas de seguridad crítica sino de cualquier tipo. Los defectos del software son difíciles de tratar y el objetivo es reducirlos al máximo posible, puesto que es prácticamente imposible erradicarlo al 100% (Miranda et al. 2014).

Es importante mencionar que estamos asumiendo que la lógica especificada es correcta y que los problemas vienen de defectos que hacen que el software no siga la lógica definida. Si tenemos problemas en cuanto a la lógica del sistema (especificaciones de lo que debe hacer) nos encontramos en un caso de errores de diseño, los cuales deberían haberse detectado y subsanado en las etapas de diseño. Por lo tanto es posible eliminar defectos del sistema verificando que el código desarrollado hace exactamente lo que se ha especificado que debe hacer. De esa forma en el uso real, cuando se den esas condiciones, hará lo que se espera de él (Lluna 2011).

El análisis estático es una evaluación del código generado para buscar defectos con la particularidad de que se realiza sin la necesidad de ejecutar el código fuente, es, por lo tanto, una técnica que se aplica directamente sobre el código fuente tal cual, sin transformaciones previas ni cambios de ningún tipo. La idea es que, en base a ese código fuente, podamos obtener información que nos permita mejorar la base de código manteniendo la semántica original (Expósito 2009), (Lluna 2011), (Miranda et al. 2014).

Las herramientas de análisis estático incluyen, analizadores léxicos y sintácticos que procesan el código fuente y, por otro lado, un conjunto de reglas que aplicar sobre determinadas estructuras. Si nuestro código fuente posee una estructura concreta que el

analizador considere como "mejorable" en base a sus reglas nos lo indicará y nos sugerirá una mejora, con éste análisis ganamos en facilidad de mantenimiento y de desarrollo ya que su objetivo es minimizar la deuda técnica de nuestros proyectos, y es que algunas de las funciones de los analizadores consisten en encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, complicar el flujo de datos, tener una excesiva complejidad, suponer un problema en la seguridad (Expósito 2009).

En cuanto a las técnicas que pueden aplicarse para realizar este análisis podemos comentar las siguientes: el análisis de valores extremos, el análisis del flujo de control y de datos y las reglas de codificación (Lluna 2011).

En la Tabla 1 se presenta en forma de resumen los aspectos más importantes de las técnicas antes mencionadas.

Técnica	Actividad
Reglas de codificación	Reglas de codificación
Análisis de valores extremos	División por cero
	Uso de objetos nulos
	Uso del valor mayor posible de una variable
	Uso del valor mínimo posible de una variable
	Acceso fuera de rango de arreglos
	Uso correcto de arreglos y listas vacías
	Comprobación de rangos de parámetros de las funciones
Análisis de flujo de control	Código accesible
	Comprobación de código anidado
	Todos los ciclos o bucles tienen condición de salida alcanzable
Análisis de flujo de datos	Inicialización de variables
	No se escriben variables más de una vez antes de ser leídas
	No se escriben variables que posteriormente no son leídas

Tabla 1 Técnicas de Análisis Estático
Elaborada por Romero Ricardo, (2016)

Sin embargo, a pesar de sus ventajas, los analizadores estáticos del código tienen las siguientes limitaciones:

- No nos permiten saber si el software va a hacer lo que se espera de él o no. Únicamente podemos analizar el código fuente y saber cómo mejorarlo.

- Al utilizar analizadores automáticos estos nos pueden devolver falsos positivos. Es posible que el analizador detecte como error algo que nosotros sabemos que está bien y que por alguna buena razón está así programado.
- Los analizadores estáticos del código están muy ligados a lenguajes de programación concretos, e incluso entre lenguajes hay diferencias, ya que en lenguajes estáticos como *Java* es más sencillo hacer un análisis estático del código que en otros lenguajes dinámicos como *groovy*.

Las características deseables de una herramienta de análisis estático son las siguientes (Lamas 2011), (Lluna 2011), (Miranda et al. 2014) :

1. Usabilidad. La herramienta debe ser fácil de instalar, de usar y de obtener resultados. Además se debe de valorar: la ayuda de la herramienta y la documentación, el tiempo de instalación, el tiempo de configuración del entorno, las actualizaciones y la facilidad para interpretar los resultados que genere la herramienta.
2. Eficiencia. Analizar el tiempo de ejecución y el consumo de recursos de cada herramienta.
3. Extensibilidad. Que la herramienta permita añadir reglas fácilmente, si la herramienta puede extenderse se podrá adaptarla a nuestro proyecto y en consecuencia no habrá límite en la cantidad de defectos que pueden ser detectados.
4. Precisión. Comparación del número de defectos comunes y que hayan sido encontrados por al menos dos analizadores de código.
5. Librerías disponibles e integración en ambientes integrados de desarrollo (*IDE*). A mayor número de *plugins* más posibilidades de utilizarla en distintas plataformas.
6. La técnica de análisis usada por la herramienta.

Las técnicas de análisis son:

- Análisis basado en *Abstract Syntax Tree* (*AST*). Este análisis no analizan el código fuente tal y como está escrito, en su lugar transforman el código en una representación de árbol (Árbol de Sintaxis Abstracta) que refleja la estructura del archivo.
- Análisis *Dataflow*. Es la técnica más utilizada en el análisis estático, es un esquema de análisis de flujo de datos define un valor en cada punto en el programa. Las instrucciones del programa tienen funciones de transferencia asociadas que relacionan el valor antes de la instrucción con el valor después de la misma. Dentro de éste análisis existen varias categorías, entre las cuales encontramos:
- Análisis Intraprocedural. Un análisis intraprocedural opera a través de un método o procedimiento.
- Análisis Interprocedural. Un análisis interprocedural opera en todo el programa entre diferentes métodos o procedimientos.
- Análisis *Context-Sensitive*. Contabiliza el número de llamadas a un procedimiento, método o función, distinguiendo dos llamadas a un mismo método.
- Análisis *Context-Insensitive*. Contabiliza el número de llamadas a un procedimiento, método o función, sin distinguir dos llamadas a un mismo método.
- Análisis *Path-Sensitive vs Path-Insensitive*. Los analizadores *path-sensitive* tienen en cuenta las distintas alternativas que podemos tener en una sentencia de decisión o bifurcación, mientras que los analizadores *path-insensitive* no tendrían en cuenta las dos alternativas, evidentemente, los primeros resultan mucho más caros de implementar que los segundos.
- Análisis *Backward Dataflow* vs Análisis *Forward Dataflow*. Estas técnicas se utilizan cuando es necesario investigar el flujo de datos en dos sentidos (*Forward* y *Backward*). La técnica *Forward* analiza el código sobre el comportamiento pasado, alcance de las definiciones o expresiones antes definidas, mientras que en la técnica *Backward* el análisis intenta predecir el comportamiento futuro, expresiones muy usadas o variables sin vida. Se pueden utilizar para descubrir redundancias en el código.

- Análisis *Style-Checkers*. Se centran exclusivamente en la estructura léxica y sintáctica. Con este tipo de analizadores se detectan espacios en blanco, código demasiado extenso, nombres de variables que no siguen las convenciones indicadas, líneas de código mal identadas.
- Análisis *Bug-Checker*. Método basado en búsqueda de patrones o reglas (*pattern matching*) predefinidos por la herramienta o diseñados por el usuario de la misma, es más utilizado que el análisis *style-checkers*.

2.4.- Servidores de IC

Existe un número bastante elevado de herramientas de IC, en base a la comparativa presentada por Katsubo (Katsubo 2015) los servidores de IC que pertenecen a la categoría de software libre y que sean construidos o que soporten el lenguaje de programación *Java* son: *CruiseControl*, *Anthill*, *LuntBuild*, *Apache Continuum*, *Hudson* y *Jenkins*.

La herramienta *LuntBuild* se omite en la comparativa, ya que la última versión data del 2009. *Anthill*, también se omite de esta comparativa, ya que la versión gratis solamente soporta la versión de *Java* 1.4 y la fecha de la última versión liberada es del 2010. *Hudson* es la base de *Jenkins*, *Hudson* desde sus inicios era libre y apoyada por la compañía de software *Sun Corporation*, *Jenkins* surge cuando la compañía de software *Oracle Corporation* en el año 2010 compra a *Sun Corporation*, la mayoría de sus desarrolladores principales y su comunidad (usuarios y colaboradores) deciden crear *Jenkins*, por lo tanto tiene un mayor soporte, una mayor comunidad y más *plugins* desarrollados que *Hudson*, en base a lo anterior también se omite la herramienta *Hudson* de la comparativa de servidores de IC.

En la Tabla 2 se generó la comparativa en donde se evalúan los servidores de IC:

Parámetros Evaluados	Apache Continuum[6]	CruiseControl[7]	Jenkins[8]
INSTALACIÓN			
Documentación sobre instalación	Suficiente	Buena	Buena, con ayuda contextual
Dificultad de instalación	Fácil. Requiere la configuración de recursos JNDI (SMTP, base de datos, etc.)	Fácil. Puede requerir configuración de archivos externos	Muy Fácil. Es posible su ejecución directamente sobre cualquier servidor de aplicaciones ligero, sin requerir editar los archivos de configuración
Configuración avanzada	Compleja	Fácil	Fácil
ADMINISTRACIÓN			
Se configura desde una interfaz Web	Sí. A excepción de algunos parámetros como el SMTP, conexión base de datos, parámetros JNDI.	Completa	Completa
Tiene documentación oficial para el administrador	Suficiente	Suficiente	Buena, con ayuda contextual
Mantiene un historial de compilaciones	Si	Si	Si
Permite gestionar tareas programadas	Si	Si	Si
Rendimiento de la herramienta	Bueno	Bueno	Muy Bueno, permite gestión de hilos de ejecución y tareas distribuidas

Tabla 2 Comparativa de servidores de IC
Elaborada por Romero Ricardo, (2016)

[6] Apache Continuum página web https://continuum.apache.org/
[7] CruiseControl página web http://cruisecontrol.sourceforge.net/
[8] Jenkins página web https://jenkins-ci.org/

Parámetros Evaluados	Apache Continuum[9]	CruiseControl[10]	Jenkins[11]
SEGURIDAD			
Gestión de permisos basados en perfiles	Si	Sí, pero sólo con un conjunto de perfiles predefinido	Si
Gestión de permisos por proyecto	Si	Si	Si
Configuración de la seguridad	Fácil	Fácil	Muy Fácil
Licencia	Apache License 2.0	BSD	MIT License
INTEGRACIÓN CON OTRAS HERRAMIENTAS			
Integración con sistemas de control de versiones	Suficiente	Suficiente	Muy Buena
Integración con herramientas de generación de reportes	Buena, basada en Maven	Buena	Muy Buena, con plugins integrados
Desarrollo de nuevos plugins o mecanismos de extensión	Complejo	Suficiente	Muy Bueno
Integración con herramientas de pruebas	Buena	Buena	Muy Buena, con agregación de los resultados en la propia interfaz web.
Soporte a proyectos Java	Muy Buena	Buena	Excelente, permite un gran número de posibilidades

Tabla 2 Comparativa de servidores de IC (Continuación)
Elaborada por Romero Ricardo, (2016)

[9] Apache Continuum página web https://continuum.apache.org/
[10] CruiseControl página web http://cruisecontrol.sourceforge.net/
[11] Jenkins página web https://jenkins-ci.org/

Parámetros Evaluados	Apache Continuum[12]	CruiseControl[13]	Jenkins[14]
FACILIDAD DE USO			
Documentación de usuario	Buena	Suficiente	Muy Buena, con ayuda contextual
Interfaz	Fácil	Simple	Muy fácil
Permite compilación distribuida	No	No	Si
Permite la gestión de hilos de ejecución	No	No	Si
Versión de la herramienta	1.4.2 (13 Junio 2014)	2.8.4 (15 Septiembre 2010)	1.633 (11 Octubre 2015)
REQUERIMIENTOS			
Versión de Java Soportada	5.0 o superior	5.0 o superior	5.0 o superior
Memoria	Sin requerimientos mínimos	Sin requerimientos mínimos	1 GB, preferible 2 GB
Espacio en disco	30 MB	No especificado	50 MB
Sistema Operativo	Windows XP, Debian, Fedora, Solaris y Mac OS X	Windows, Unix	Windows, Ubuntu, Debian, Red Hat, Fedora, CentOS, Mac OS X, openSUSE, FreeBSD, OpenBSD y Gentoo

Tabla 2 Comparativa de servidores de IC (Continuación)
Elaborada por Romero Ricardo, (2016)

Como se puede observar, Jenkins reúne las principales características que esperamos de un sistema de IC (extensibilidad, calidad en la documentación, seguridad, estabilidad, soporte de la herramienta, entre otras características), y por ello es nuestra propuesta y elección para servir de servidor de IC en el desarrollo del presente trabajo.

[12] Apache Continuum página web https://continuum.apache.org/
[13] CruiseControl página web http://cruisecontrol.sourceforge.net/
[14] Jenkins página web https://jenkins-ci.org/

2.5.- Metodología orientada a objetos

Estas metodologías crean una representación o abstracción del mundo real en el ámbito del software, se fueron convirtiendo en el estilo de programación dominante desde mediados de los años 1980, debido a la influencia del lenguaje de programación *C++*. Posteriormente se consolidado gracias al auge de las interfaces gráficas de usuario, para las cuales éstas metodologías están bien adaptadas.

Meyer (Meyer 1999) afirma que el enfoque orientado a objetos abarca todo el ciclo de desarrollo de software, ya que se aplican en el análisis (Análisis Orientado a Objetos), diseño (Diseño Orientado a Objetos), implementación y mantenimiento del software (Programación Orientada a Objetos). Existe un único mecanismo computacional básico que es el objeto, el cuál es siempre una instancia de una clase.

Por otro lado, Pressman (Pressman 2010) sostiene que en el contexto de la ingeniería de software orientada a objetos, un componente contiene un conjunto de clases que colaboran, en donde cada clase dentro de un componente se elabora para que incluya todos los atributos y operaciones relevantes para su implantación. La ingeniería de software orientada a objetos comienza con el modelo de requerimientos y se elaboran clases de análisis (para los componentes que se relacionan con el dominio del problema) y clases de infraestructura (para los componentes que dan servicios de apoyo para el dominio del problema).

El Análisis Orientado a Objetos (AOO) es un método de análisis para examinar los requerimientos desde una perspectiva de clases y objetos, buscados en el vocabulario del dominio del problema, para poder representar la experiencia del usuario en la esquematización del requerimiento.

El Diseño Orientado a Objetos (DOO) es un método de diseño para comprender el proceso de descomposición y notación orientada a objetos, obteniendo el modelo lógico (estructuras de clases y objetos) y físico (arquitectura de módulos y procesos), así como los modelos estáticos y dinámicos.

La Programación Orientada a Objetos (POO) es la implementación del diseño, en donde los programas son colecciones de objetos cooperantes. Cada objeto representa una instancia de alguna clase y las clases pertenecen a una jerarquía de clases relacionadas por la herencia.

Factores de Calidad

Las metodologías orientadas a objetos tienen su origen en la investigación sobre los métodos conceptuales que deben introducirse para mejorar la calidad del software que se desarrolla. Los factores de calidad que son mejorados por la orientación a objetos son (Meyer 1999), (Pressman 2010), (Schach 2005):

Correctitud

Es la capacidad de una aplicación software para operar de acuerdo a como está definida en su especificación; grado en el que una aplicación de software satisface sus expectativas y en el que cumple con los objetivos de la misión del cliente.

Reusabilidad

Es la adecuación de un producto software para poder ser reusado en otras aplicaciones para las que no fue diseñado; grado en el que una aplicación de software (o partes) pueden volverse a utilizar en otras aplicaciones (se relaciona con el empaquetamiento y alcance de las funciones que realicen).

Extensibilidad

Es la facilidad que presenta un producto de software para ser adaptado tras un cambio de las especificaciones, las dos bases de la extensibilidad son: la simplicidad del diseño y el diseño descentralizado; capacidad de una aplicación de software para ser ampliable.

Robustez

Es la capacidad de un producto de software para dar una respuesta satisfactoria bajo situaciones no establecidas en la especificación; grado en el que el software maneja entradas erróneas de datos o en el que se presenta una interacción inapropiada por parte del usuario.

Compatibilidad

Es la facilidad que presenta un producto software para poder ser combinado con otros, las claves para conseguir la compatibilidad son: la homogeneidad y la estandarización; grado en el que el producto de software funcione correctamente en diferentes ambientes (distinto hardware, sistemas operativos, navegadores, entre otros).

Características del paradigma orientado a objetos

La orientación a objetos es un método que se basa en las interacciones entre objetos para resolver las necesidades de una aplicación de software. Las características principales son (Meyer 1999), (Pressman 2010), (Schach 2005):

Encapsulación

Define de forma independiente la abstracción o interfaz y su implementación y estructura interna; la encapsulación es el proceso de compartir elementos de una abstracción que constituyen su estructura y funcionamiento. La encapsulación sirve para separar la interface de una abstracción y su implementación. Esta característica es evaluada por las métricas MOOD (*MHF y AHF*) tal como se muestra en la Tabla 3.

Herencia

La herencia es la capacidad de agrupar jerárquicamente los componentes de forma que aquellas características que tengan en común muchos de ellos solo se necesite describir una vez a través de la descripción del correspondiente antecesor común; jerarquiza las clases de acuerdo con afinidades de sus abstracciones. La herencia es evaluada por las métricas CK (*DIT y NOC*) y MOOD (*MIF y AIF*) como se presenta en la Tabla 3.

Polimorfismo

Es la capacidad de una entidad consistente en poder conectarse a objetos de varios tipos, no puede ser arbitrario y debe de estar controlado por la herencia; es un concepto, en donde el nombre de un método, puede denotar operaciones de objetos de varias clases, siempre y

cuando estas se relacionen por alguna superclase común, es muy útil cuando tenemos muchas clases con el mismo protocolo. Con el polimorfismo, se eliminan grandes instrucciones de tipo case, porque cada objeto conoce su tipo y por lo tanto sabe cómo reaccionar. Esta característica es evaluada por las métricas MOOD (*PF y CF*), como se muestra en la Tabla 3.

Modularización

La modularización consiste en descomponer un componente complejo en un conjunto reducido de subcomponentes más sencillos. La estrategia de modularización debe iterarse hasta que la complejidad de los módulos que resultan sea abordables por el programador; describe el sistema como conjunto de módulos (objetos) descentralizados y débilmente acoplados. La modularización es evaluada por las métricas CK (*WMC, RFC, MPC y DAC*), como se presenta en la Tabla 3.

Abstracción

La abstracción es la capacidad reducir la información que describe un componente a la necesaria para manejarlo dentro de la fase de desarrollo; busca una definición conceptual común a muchos objetos, tratando de identificar sus características esenciales y agrupándolos por clases. Esta característica es evaluada por las métricas CK (*DAC*), como se puede observar en la Tabla 3.

Acoplamiento

El acoplamiento es la medida de fortaleza en la asociación establecida por una conexión de un módulo a otro, o bien entre clases. Un fuerte acoplamiento evita la independencia entre los módulos o clases del sistema; un módulo o clase fuertemente acoplado con otro es rígido de entender, cambiar o corregirse, por lo tanto, la complejidad de un diseño de sistema puede ser reducido utilizando un débil acoplamiento entre módulos o clases. El acoplamiento es evaluada por las métricas CK (*CBO y MPC*) y MOOD (*CF*), tal como lo muestra la Tabla 3.

Cohesión

La cohesión es la medida del grado de colectividad entre los elementos de un diseño orientado a objetos como un módulo, objetos simples o clases simples; la cohesión no tiene tanto impacto sobre la modularidad como el acoplamiento. Es decir, un gran acoplamiento puede ser muy malo a la hora de corregir errores, integrar partes, hacer mantenimientos. La cohesión es evaluada por las métricas CK (*LCOM, LCOM1 y LCOM2*), como se presenta en la Tabla 3.

2.6.- Métricas de calidad

2.6.1.- Métricas de Chidamber y Kemerer (CK)

Shyam R. Chidamber y Chris F. Kemerer desde 1994 establecieron un total de seis métricas para la medición de la calidad en el desarrollo de aplicaciones orientadas a objetos: *WMC (Weighted Methods Per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling Between Object Classes), RFC (Response for Class) y LCOM (Lack of Cohesion of Methods)*. Estas métricas no están enfocadas a la productividad, sino a la calidad, ya que tratan de evaluar de manera cuantitativa, si ciertas propiedades deseables del diseño orientado a objetos, se cumplen, tales como la complejidad, el acoplamiento, la cohesión, la herencia y la comunicación entre clases (Greiner et al. 2010), (Cheikhi et al. 2014), (Ma et al. 2010).

Weighted Methods Per Class (WMC)

Los métodos ponderados por clase (*WMC*) mide la complejidad de una clase. Clases con un gran número de métodos requieren más tiempo y esfuerzo para desarrollarlas y mantenerlas, ya que influirán en las subclases que heredan todos sus métodos. Además, estas clases tienden a ser específicas del sistema, limitando su posibilidad de reúso.

Definición:

Considere una clase C_i con los métodos $M_1, M_2, \ldots, M_n$.

Sean $c_1, c_2, \ldots, c_n$ la WMC de los métodos, entonces:

$$WMC = \sum_{i=1}^{n} c_i$$

Categoría de la Métrica: Complejidad.

Cheikhi et al (Cheikhi et al. 2014) proponen la siguiente base teórica para la métrica *WMC*: *WMC* se relaciona directamente con la definición de complejidad de un objeto, ya que los métodos son propiedades de los objetos y la complejidad de un objeto está determinado por la cardinalidad de su conjunto de propiedades. El número de métodos es, por lo tanto, una medida de definición de objeto, así como atributos de un objeto, ya que los atributos corresponden a propiedades.

Datos de Referencia: El rango óptimo *WMC* es de 20 a 25 (Misra & Bhavsar 2003).

Depth of Inheritance Tree (DIT)

La profundidad en el árbol de herencia (*DIT*) mide el máximo nivel en la jerarquía de herencia. La *DIT* es la cuenta de los niveles en la jerarquía de herencia. En el nivel cero de la jerarquía se localizan la clase raíz. Esta métrica representa la medida de la complejidad de una clase: la complejidad del diseño y el potencial reúso de la misma, esto se debe a que cuanto más profunda se encuentra una clase en la jerarquía, mayor será la probabilidad de heredar un mayor número de métodos. El uso de la herencia es visto como un compromiso ya que:

- Altos niveles de herencia indican objetos complejos, los cuales pueden ser difíciles de probar y reusar.
- Bajos niveles en la herencia pueden indicar que el código está escrito en un estilo funcional sin aprovechar el mecanismo de herencia proporcionado por la orientación a objetos.

Definición:

La métrica DIT de una clase A es su profundidad en el árbol de herencia.

Si A se encuentra en situación de herencia múltiple,

la longitud máxima hasta la raíz será el DIT.

Categoría de la Métrica: Herencia.

Cheikhi et al (Cheikhi et al. 2014) proponen la siguiente base teórica para la métrica *DIT*: *DIT* se refiere a la noción de alcance de propiedades. *DIT* es una medida de la cantidad de clases superiores (padres) pueden afectar potencialmente a esta clase.

Datos de Referencia: El valor mínimo es 1 y el valor recomendado es 5 o menor (Misra & Bhavsar 2003).

Number of Children (NOC)

El número de hijos (*NOC*) es el número de subclases subordinadas a una clase en la jerarquía de herencia, es decir, el número de subclases que pertenecen a una clase. El *NOC* es un indicador del nivel de reúso, la posibilidad de haber creado abstracciones erróneas y es un indicador del nivel de prueba requerido.

Es un potencial indicador de la influencia que una clase puede tener sobre el diseño del sistema. Si el diseño tiene una alta dependencia en la reusabilidad a través de herencia, puede ser mejor dividir la funcionalidad en varias clases.

Definición:

El NOC de una clase es el número de subclases

subordinadas a una clase en la jerarquía.

Categoría de la Métrica: Herencia.

Cheikhi et al (Cheikhi et al. 2014) proponen la siguiente base teórica para la métrica *NOC*: *NOC* se refiere al alcance de las propiedades de una clase. Es una medida de la cantidad de subclases que van a heredar los métodos de la clase padre.

Datos de Referencia: Valor recomendado es 5 o menor (Misra & Bhavsar 2003)

Coupling Between Object Classes (CBO)

El acoplamiento entre objetos (*CBO*) de una clase es el número de clases con las que está acoplada. Una clase está acoplada a otra si utiliza sus métodos o variables de instancia, excluyendo el acoplamiento por herencia.

Definición:

El CBO de una clase, es el número de clases con la que está acoplada.

Una clase está acoplada a otra, si utiliza sus métodos y

variables de instancia, excluyendo el acoplamiento por herencia.

Categoría de la Métrica: Acoplamiento.

Cheikhi et al (Cheikhi et al. 2014) proponen la siguiente base teórica para la métrica *CBO*: *CBO* se refiere a la idea de que un objeto está acoplado a otro objeto si dos objetos actúan uno sobre el otro, es decir, los métodos de métodos de uso de uno o variables de instancia de otro.

Datos de Referencia: Valor recomendado es entre 1 y 4 (Misra & Bhavsar 2003).

Response for Class (RFC)

La respuesta de una clase (*RFC*) es el cardinal del conjunto de todos los métodos que pueden ser invocados en respuesta a un mensaje a un objeto de la clase o por algún método en la clase. Esto incluye todos los métodos accesibles dentro de la jerarquía de la clase. En otras palabras, cuenta las ocurrencias de llamadas a otras clases desde una clase particular. La *RFC* es una medida de la complejidad de una clase a través del número de métodos y de la comunicación con otras clases, ya que incluye todos los métodos llamados desde fuera de la clase. Es un indicador de los recursos necesarios para pruebas y depuración. Cuanto mayor es *RFC*, más complejidad tiene el sistema ya que se puede invocar un mayor número de métodos como repuesta a un mensaje.

Definición:

El *RFC* de una clase es el conjunto de métodos que se pueden ejecutar como respuesta a un mensaje recibido por un objeto de la clase.

$$RFC = |RS|$$

RS es la RFC de la clase, dado que $RS = \{M\} \cup_{all\,i} \{R_i\}$, donde

$\{R_i\}$ es el conjunto de métodos invocados por el método i y

$\{M\}$ es el conjunto de todos los métodos de la clase.

Categoría de la Métrica: Colaboración.

Cheikhi et al (Cheikhi et al. 2014) proponen la siguiente base teórica para la métrica RFC: RFC es un conjunto de métodos disponibles en el objeto, y su cardinalidad es una medida de los atributos de un objeto; ya que incluye específicamente métodos llamados desde fuera del objeto, sino que también es una medida de la comunicación entre objetos.

Datos de Referencia: Valor óptimo es 15 o menor (Misra & Bhavsar 2003).

Lack of Cohesion of Methods (LCOM)

La falta de cohesión en los métodos (*LCOM*) establece en qué medida los métodos hacen referencia a atributos. La *LCOM* es una medida de la cohesión de una clase teniendo en cuenta el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase. Un valor alto de *LCOM* implica falta de cohesión, es decir, escasa similitud de los métodos. Esto puede indicar que la clase está compuesta de elementos no relacionados, incrementando la complejidad y la probabilidad de errores durante el desarrollo. Es deseable una alta cohesión en los métodos dentro de una clase lo que implica que ésta no pueda ser dividida, lo cual fomenta la encapsulación.

Definición:

Considere una clase C_i con métodos M_1, M_2, …, M_n.

Sea $\{I_j\}$ el conjunto de variables de instancia usados por el método M_i.

Existen n de esos conjuntos $\{I_1, I_2, \ldots, I_n\}$,

sea $P=\{(I_i, I_j) \vee I_i \cap I_j = 0\}$ y $Q=\{(I_i, I_j) \vee I_i \cap I_j \neq 0\}$ entonces:

$$LCOM = |P| - |Q|, si |P| > |Q|; de\ otra\ forma\ LCOM = 0.$$

Categoría de la Métrica: Cohesión.

Cheikhi et al (Cheikhi et al. 2014) proponen la siguiente base teórica para la métrica *LCOM*: *LCOM* utiliza la noción de grado de similitud entre métodos. Si no hay variables de instancia comunes, el grado de similitud es cero. Sin embargo, esto no distingue entre el caso en el que cada uno de los métodos funciona con un conjunto único de las variables de instancia y el caso en que un solo método funciona en un único conjunto de variables. *LCOM* está íntimamente ligada a las variables y los métodos de un objeto de instancia, y por lo tanto es una medida de los atributos de un objeto.

Datos de Referencia: El valor deseable de *LCOM* es 0, mientras que si *LCOM* > 0 esto sugiere que la clase sea dividida en 2 o más clases.

Partiendo de las métricas de CK, se han ido proponiendo modificaciones y ampliaciones, Li y Henry modificaron la métrica *LCOM*, ya que consideran que la definición original es ambigua porque permite que dos clases con distintos grados de cohesión obtengan idénticos valores de *LCOM* y añaden cuatro métricas (Olmedilla 2005).

Lack of Cohesion of Methods1 (LCOM1)

Definición:

LCOM1 = Número de conjuntos disjuntos de métodos locales en una clase.

Cada conjunto tiene uno o más métodos locales de una clase, y

cualquiera de los métodos en el acceso conjunto al menos

un atributo de la clase común; el número de atributos comunes

que van de 0 a N (donde N es un número entero mayor que 0).

Las cuatro métricas propuestas tienen por objetivo medir y mejorar el mantenimiento del software orientado a objetos, añadiendo la abstracción y el tamaño del diseño (Li & Henry 1993)

Las cuatro métricas añadidas por Li y Henry son:

Message Passing Coupling (MPC)

Esta métrica establece el número de métodos que son invocados en una clase. Se usa para medir la complejidad del paso de mensajes entre clases.

Definición:

MPC = Número de métodos invocados en una clase.

Categoría de la Métrica: Acoplamiento/Comunicación.

Data Abstraction Coupling (DAC)

Esta métrica permite obtener el número de atributos de una clase que tienen como tipo a otra clase. Como una clase puede verse como una implementación de tipos de datos abstractos, esto causa un tipo particular de acoplamiento. Este tipo de acoplamiento puede violar la propiedad de encapsulación si el lenguaje de programación permite el acceso directo a propiedades privadas del tipo de datos abstractos, por lo que esta métrica mide acoplamiento complejo causado por el acoplamiento de abstracción de datos.

Definición:

DAC = Número de atributos de una clase que tiene como tipo a otra clase.

Categoría de la Métrica: Acoplamiento/Abstracción.

Datos de Referencia: El valor máximo deseable es 5.

SIZE1

Esta métrica es una variación de la métrica LOC (Número de líneas de código), la cual es definida exclusivamente para el lenguaje Ada, por lo que no se tomará en cuenta en el presente trabajo.

SIZE2

Esta métrica define el número de propiedades (incluidos los atributos y métodos) definidos en una clase.

Definición:

SIZE2 = Número de atributos + número de métodos de una clase.

Categoría de la Métrica: Tamaño del diseño.

Lack of Cohesion of Methods2 (LCOM2)

Henderson-Sellers proponen otra variante de la métrica *LCOM*, la cual definen como una normalización del número de métodos y variables de una clase (Badri et al. 2011).

Definición:

LCOM2 = (a – kl) / (l – kl).

l = Número de atributos.

k = Número de métodos.

a = Sumatoria de los distintos atributo accedidos por cada método.

Datos de Referencia: El resultado se da en un rango entre 0 y 1, donde el valor deseable de *LCOM2* es 0.

2.6.2.- Métricas para el diseño orientado a objetos (MOOD)

Estas métricas (a diferencia de las métricas CK que realizan la evaluación a nivel de clase), consideran distintos niveles de granularidad, realizan la evaluación de todo el sistema de acuerdo a la herencia, acoplamiento, polimorfismo y encapsulamiento, ésta diferencia hace que éstas métricas están enfocadas a la productividad (Greiner et al. 2010), (Ma et al. 2010).

Polymorphism Factor (PF)

El polimorfismo es una característica importante en el paradigma orientado a objetos. El polimorfismo mide el grado de sobre escritura en una estructura jerárquica de clases.

El factor de polimorfismo (*PF*) representa el número real de posibles situaciones polimórficas diferentes, también representa el número máximo de posibles situaciones polimórficas diferentes para una clase en particular. El *PF* es además, una medida indirecta de la asociación dinámica del sistema.

Definición:

$$PF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) * DC(C_i)]}$$

Donde:

$M_0(C_i) = M_n(C_i) + M_0(C_i)$

$M_n(C_i)$ = Número de métodos definidos en la clase C_i

$M_0(C_i)$ = Número de métodos sobrecargados de la clase C_i

$DC(C_i)$ = Número de descendientes (hijos) de la clase C_i

TC = Total de clases

Categoría de la Métrica: Polimorfismo.

Datos de Referencia: El rango aceptable es del 0% al 10% (Brito 2005).

Coupling Factor (CF)

El factor de acoplamiento (*CF*) se evalúa como una fracción, donde el numerador representa el número de acoplamientos no relacionados con la herencia y el denominador es el número máximo de acoplamientos en un sistema. El número máximo de acoplamientos incluye tanto la herencia y el acoplamiento no relacionada con la herencia. Los acoplamientos basados en herencia surgen como clases derivadas (subclases) y de la herencia de métodos y atributos forman su clase padre (superclase).

Definición:

$$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_{client}(C_I, C_J) \right]}{TC}$$

Donde:

$is_{client}(C_i, C_j)$ = la relación entre la clase cliente y la clase objeto

TC = total de clases

Categoría de la Métrica: Acoplamiento.

Datos de Referencia: El valor recomendado es menor del 10% (Brito 2005).

Method Hiding Factor (MHF)

El factor de ocultamiento de métodos (*MHF*) mide la invisibilidad de los métodos en las clases. La invisibilidad de un método es el porcentaje de las clases totales de la que el método no es visible. Esta métrica es una fracción donde el numerador es la suma de todos los métodos ocultos definidos en el sistema. El denominador es el número total de los métodos definidos en el sistema. Los métodos deben ser encapsulados (ocultos) dentro de una clase y no está disponible para su uso a otros objetos. Un método oculto aumenta la reutilización de las clases en otras aplicaciones y reduce la complejidad. Si hay una necesidad de cambiar la funcionalidad de un método en particular, las acciones correctivas se tienen que tomarse en todos los objetos de acceso a ese método, si el método no está oculto. Por lo tanto los métodos ocultos también reducen las modificaciones en el código. El *MHF* debería tener un valor muy grande.

Definición:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Donde:

$M_d(C_i) = M_v(C_i) + M_h(C_i)$

$M_d(C_i)$ = Número de métodos definidos en la clase C_i

$M_v(C_i)$ = Número de métodos visibles de la clase C_i

$M_h(C_i)$ = Número de métodos ocultos de la clase C_i

TC = Total de clases

Categoría de la Métrica: Encapsulación.

Datos de Referencia: El rango aceptable es del 12% al 22% (Brito 2005).

Attribute Hiding Factor (AHF)

El factor de ocultamiento de atributos (*AHF*) mide la invisibilidad de los atributos de las clases. La invisibilidad de un atributo es el porcentaje de las clases totales de la que el atributo no es visible. Un atributo se llama visible si se puede acceder por otra clase u objeto. Los atributos deben ser ocultados en una clase. Se pueden conservar el acceso de otros objetos al ser declarada como privado. El *AHF* es una fracción. El numerador es la suma de todos los atributos no visibles definidos en todas las clases. El denominador es el número total de atributos definidos en el sistema. Es deseable que el atributo de ocultación factor a tener un valor grande.

Definición:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Donde:

$A_d(C_i) = A_v(C_i) + A_h(C_i)$

$A_d(C_i)$ = Número de atributos definidos en la clase C_i

$A_v(C_i)$ = Número de atributos visibles de la clase C_i

$A_h(C_i)$ = Número de atributos ocultos de la clase C_i

TC = Total de clases

Categoría de la Métrica: Encapsulación.

Datos de Referencia: El valor aceptable es mayor del 75% (Brito 2005).

Method Inheritance Factor (MIF)

La herencia es el proceso por el cual los objetos de una clase adquieren las propiedades de los objetos de otra clase. Las características heredadas de una clase son aquellas que hereda de otra clase (clase padre) y no son las que sobrescribe. El factor de herencia de métodos (*MIF*) es la suma de los métodos heredados en todas las clases del sistema. Mide el grado en que una clase en una arquitectura orientada a objetos hace uso de la herencia de métodos y atributos.

Definición:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Donde:

$M_a(C_i) = M_d(C_i) + M_i(C_i)$

$M_a(C_i)$ = Número de métodos definidos en la clase C_i

$M_d(C_i)$ = Número de métodos declarados de la clase C_i

$M_i(C_i)$ = Número de métodos heredados de la clase C_i

TC = Total de clases

Categoría de la Métrica: Herencia.

Datos de Referencia: El rango aceptable es del 60% al 80% (Brito 2005).

Attribute Inheritance Factor (AIF)

El factor de herencia de atributos (*AIF*) se define como la relación de la suma de atributos heredados en todas las clases del sistema entre el número total de atributos disponibles para todas las clases. El *AIF* expresa el nivel de reutilización en un sistema.

Definición:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Donde:

Aa(Ci) = Ad(Ci) + Ai (Ci)

Aa(Ci) = Número de atributos definidos en la clase Ci

Ad(Ci) = Número de atributos declarados de la clase Ci

Ai (Ci) = Número de atributos heredados de la clase Ci

TC = Total de clases

Categoría de la Métrica: Herencia.

Datos de Referencia: El rango aceptable es del 50% al 70% (Brito 2005).

2.6.3.- Resumen de métricas

En la Tabla 3 se muestran las métricas que se proponen verificar en proyectos de software implementados en lenguaje *Java*, a dichas métricas se les ha agregado el atributo de calidad a las que pertenecen.

Tipo	Métrica	Propiedad DOO	Valor Permitido	Atributo de Calidad
CK	WMC	Complejidad	20 al 25	Fiabilidad, Escalabilidad, Desempeño, Integración
CK	DIT	Herencia	1 al 5	Funcionalidad, Eficiencia, Mantenibilidad, Desempeño, Escalabilidad, Integración
CK	NOC	Herencia	5	Funcionalidad, Eficiencia, Mantenibilidad, Desempeño, Escalabilidad, Integración
CK	CBO	Acoplamiento	1 a 4	Fiabilidad, Escalabilidad, Mantenibilidad, Integración
CK	RFC	Colaboración, Comunicación	15	Fiabilidad, Escalabilidad, Mantenibilidad, Integración, Eficiencia
CK	LCOM	Cohesión	deseable 0	Fiabilidad, Escalabilidad, Mantenibilidad, Integración
CK*	LCOM1	Cohesión	deseable 0	Fiabilidad, Escalabilidad, Mantenibilidad, Integración
CK*	LCOM2	Cohesión	0 a 1, deseable 0	Fiabilidad, Escalabilidad, Mantenibilidad, Integración
CK*	MPC	Acoplamiento, Comunicación		Fiabilidad, Escalabilidad, Mantenibilidad, Integración, Eficiencia
CK*	DAC	Acoplamiento, Abstracción	5	Fiabilidad, Escalabilidad, Mantenibilidad, Integración, Eficiencia
CK*	SIZE2	Tamaño del diseño		Mantenibilidad, Eficiencia, Escalabilidad
MOOD	PF	Polimorfismo	0 al 10%	Fiabilidad, Escalabilidad, Mantenibilidad, Integración, Eficiencia
MOOD	CF	Acoplamiento	< 10%	Fiabilidad, Escalabilidad, Mantenibilidad, Integración
MOOD	MHF	Encapsulación	12 a 22%	Eficiencia, Mantenibilidad, Escalabilidad
MOOD	AHF	Encapsulación	> 75%	Eficiencia, Mantenibilidad, Escalabilidad
MOOD	MIF	Herencia	60 al 80%	Funcionalidad, Eficiencia, Mantenibilidad, Desempeño, Escalabilidad, Integración
MOOD	AIF	Herencia	50 al 70%	Funcionalidad, Eficiencia, Mantenibilidad, Desempeño, Escalabilidad, Integración

Tabla 3 Resumen de Métricas
Elaborada por Romero Ricardo, (2016)

3.- ANALISIS DE HERRAMIENTAS

3.1.- Evaluación de las herramientas de análisis estático del software

En base a las características deseables de la herramienta de análisis estático citadas por los autores Lluna (Lluna 2011), Miranda et al (Miranda et al. 2014), Lamas (Lamas 2011) y Rutar et al (Rutar et al. 2004) se presenta la Tabla 4 donde son evaluadas las herramientas *Java* más populares para el análisis estático.

Parámetros Evaluados	FindBugs[15]	PMD[16]	CheckStyle[17]
USABILIDAD			
Documentación	Documentación en línea, muy básica, actualizada	Documentación en línea, actualizada	Documentación en línea, actualizada y muy detallada
Ayuda	Suficiente, el sitio no tiene ayuda para la construcción de nuevas reglas	Buena, muy completa	Muy buena, muy completa y con ayuda contextual
Tiempo de instalación	5 minutos, instalación muy sencilla, con pocos pasos	5 minutos, instalación muy sencilla, con pocos pasos	5 minutos, instalación muy sencilla, con pocos pasos
Actualizaciones	Se actualiza constantemente	Se actualiza constantemente	Se actualiza constantemente
Facilidad de interpretar los resultados	Permite exportar los resultados a *XML*, organiza el reporte de incidencias en varias categorías según su gravedad, a cada reporte se le asigna una prioridad: alta, media y baja. En condiciones normales no reporta los fallos de prioridad baja	Por default genera un archivo *RTF*; el informe de resultados depende de la librerías utilizada, Los fallos están clasificados en 5 niveles: *high error, error, high warning, warning e information*; incorpora además una funcionalidad para generar estadísticas resumen; los resultados aparece más documentados	Permite exportar los resultados a varios formatos de archivos, permite interactuar con otras herramientas para presentar los resultados en una página *web*, con filtros y niveles de consulta.

Tabla 4 Comparativa de herramientas Java de análisis estático
Elaborada por Romero Ricardo, (2016)

[15] FindBugs página web http://findbugs.sourceforge.net/
[16] PMD página web https://pmd.github.io/
[17] CheckStyle página web http://checkstyle.sourceforge.net/

Parámetros Evaluados	FindBugs[18]	PMD[19]	CheckStyle[20]
EFICIENCIA			
Tiempo de ejecución	Variable, depende de la forma de ejecución. Para ejecutarla podemos utilizar *Ant, Maven, Swing GUI*, desde la línea de comandos o integrado en *IDE's* (*Eclipse o Netbeans*), es lento	Es más rápido y localiza muchos más defectos que *FindBugs*	Variable, depende de la forma de ejecución. Para ejecutarla podemos utilizar *Ant, Maven*, desde la línea de comandos o integrado en *IDE's* (*Eclipse, IntelliJ, NetBeans, jEdit, tIDE, JDeveloper, JBuilder, BlueJ*)
Consumo de recursos	Consume más recursos y se hace necesario cambiar la configuración por default; añade en los archivos *XML* atributos de tiempo de ejecución para cada archivo y proyecto haciéndolo más lento	No se dispone de este tipo de información	No se dispone de este tipo de información
EXTENSIBILIDAD			
Permite añadir reglas fácilmente	Si, el proceso es un poco largo	Trae definidos un conjunto de reglas agrupado en *ruleset* temáticos, proceso algo largo	Altamente configurable y puede soportar casi cualquier estándar de codificación, se configura por medio de un archivo *XML*; usa los estándares *Google Java Style Guide* y *Sun Code Conventions* por default; permite seleccionar sobre qué convenciones se quiere generar la alarma y sobre cuáles no; encuentra fallos en la documentación *Javadoc*
Existen editores especiales	No, cualquier procesador de textos se puede usar	Usa editor propio; se puede extender con el módulo *PMD Rule Designer*	No, cualquier procesador de textos se puede usar
LIBRERIAS			
Librerías disponibles	*Eclipse, Netbeans*	*Eclipse, IntelliJ, NetBeans, jEdit, JBuilder, BlueJ*	*Eclipse, IntelliJ, NetBeans, jEdit, tIDE, JDeveloper, JBuilder, BlueJ*

Tabla 4 Comparativa de herramientas Java de análisis estático (Continuación)
Elaborada por Romero Ricardo, (2016)

[18] FindBugs página web http://findbugs.sourceforge.net/
[19] PMD página web https://pmd.github.io/
[20] CheckStyle página web http://checkstyle.sourceforge.net/

Parámetros Evaluados	FindBugs[21]	PMD[22]	CheckStyle[23]
TÉCNICAS DE ANÁLISIS			
Técnica usada por la herramienta	Árbol de Sintaxis Abstracta (*AST*), usa la librería *BCEL (Apache Byte Code Engineering Library)*; análisis de flujo de datos intraprocedural; análisis *bug-checker* basado en *bug pattern*	Árbol de Sintaxis Abstracta (*AST*); análisis *style checkers; bug-checker*	Árbol de Sintaxis Abstracta (*AST*), usa *ANTLR* para generarlo; análisis *style checkers*
Limitaciones	No proporciona *path-sensitive analysis*, ni un mecanismo estándar de análisis metadatos, no realiza un análisis sensitivo de contexto interprocedural, el análisis que realiza no incluye la documentación *Javadoc*	No realiza análisis de flujo de datos (*dataflow*); No soporta análisis metadata; No puede ejecutarse en más de un archivo al mismo tiempo; No analiza el código fuente del programa de manera directa, sino que transforma el código fuente en una representación en árbol (*AST*) que refleja la estructura del archivo	No analiza el código fuente del programa de manera directa, sino que transforma el código fuente en una representación en árbol (*AST*) que refleja la estructura del archivo; No hace un análisis del flujo de datos (*dataflow*)
REQUERIMIENTOS			
Versión de Java Soportada	1.5 o superior	1.6 o superior	1.5 o superior
Memoria	512 MB mínimo	Sin requerimientos mínimos	Sin requerimientos mínimos
Sistema Operativo	Windows, Linux, y Mac OS X	Windows, Linux/Unix	Windows, Linux/Unix, y Mac OS X
Licencia	LGPL (Lesser GNU Public License)	BSD (Berkeley Software Distribution)	GNU (Lesser General Public License)
Versión	3.0.1 (6 de marzo 2015)	5.4.0 (4 de octubre 2015)	6.12.1 (5 de noviembre 2015)
Número de descargas	557,481 (19 de Noviembre 2015)	748,736 (19 de Noviembre 2015)	846,904 (19 de Noviembre 2015)

Tabla 4 Comparativa de herramientas Java de análisis estático (Continuación)
Elaborada por Romero Ricardo, (2016)

[21] FindBugs página web http://findbugs.sourceforge.net/
[22] PMD página web https://pmd.github.io/
[23] CheckStyle página web http://checkstyle.sourceforge.net/

Como se puede ver en la tabla anterior, cada herramienta de análisis estático tiene aspectos y/o reglas en común, finalidad propia, fortalezas y debilidades, por lo que cada herramienta ofrece un servicio único; la selección de la herramienta a usar dependerá del enfoque que se le dé al análisis a realizar y de los recursos informáticos con los que se disponga.

3.2.- Análisis de la arquitectura del servidor de IC

La aplicación llamada Jenkins es un sistema de IC, escrita en lenguaje *Java*, de tipo web y pertenece a las herramientas de código abierto, está basado en el proyecto *Hudson*; fue desarrollado por Kohsuke Kawaguchi (Berg 2015). En noviembre de 2010 surgieron varios problemas respecto a la administración y gestión del proyecto por parte de *Oracle Corporation* en relación al proyecto *Hudson*, debido a la compra que realizó ésta de la empresa *Sun Corporation*. Uno de los puntos claves fue la propiedad de la marca *Hudson*. En diciembre de 2010 *Oracle Corporation* reclamo el derecho del nombre y la marca registrada *Hudson* y como resultado surge *Jenkins*. En la actualidad *Jenkins* y *Hudson* continúan como proyectos independientes (Ferguson 2011).

Jenkins proporciona IC para el desarrollo de software, es decir, permite configurar y agendar la ejecución de tareas conteniendo toda y/o cada una de las partes que conforman el ciclo de vida de un proyecto.

Técnicamente *Jenkins* es una aplicación web extendible, la plataforma básica de extensión es por medio de *plugins*, los cuales son los responsables de realizar la mayoría de funcionalidades de la aplicación.

Un *plugin* es una aplicación de software que añade funcionalidad a otra aplicación; lo habitual es que el *plugin* sea ejecutado desde la aplicación principal, y dicha aplicación interactúa con el *plugin* por medio de interfaces. En la actualidad, una gran mayoría de aplicaciones trabajan agregando funcionalidad por medio de *plugins*.

Jenkins utiliza una arquitectura maestro/esclavo para la administración de ejecución de tareas distribuidas. El trabajo del servidor maestro es el de administrar la programación de las tareas de construcción, enviar las versiones de código a los servidores esclavos para su ejecución

en tiempo real, dar seguimiento de los servidores esclavos y el registro y la presentación de los resultados de compilación. Mientras que el trabajo de los servidores esclavos es realizar la ejecución de construir las tareas enviados por el servidor maestro.

Un servidor esclavo es un pequeño ejecutable *Java* que se ejecuta en una máquina remota y escucha las peticiones de la instancia principal de Jenkins (servidor maestro). Los servidores esclavos funcionan en una gran variedad de sistemas operativos. La instancia esclavo se puede iniciar en un número de maneras diferentes, dependiendo del sistema operativo y arquitectura de red. Una vez que se ejecuta la instancia de esclavos, se comunica con la instancia principal través de una conexión *TCP / IP (Transmission Control Protocol / Internet Protocol)* (Berg 2015), (Ferguson 2011).

De manera muy general, se presenta en la Ilustración 4, la arquitectura básica del servidor de IC llamado *Jenkins*; esta arquitectura es compuesta por tres capas que son la capa del modelo, la capa de vista y la capa de servicios, *Jenkins* permite la ejecución de *plugins* por medio de puntos de extensión.

3.2.1.- Arquitectura General

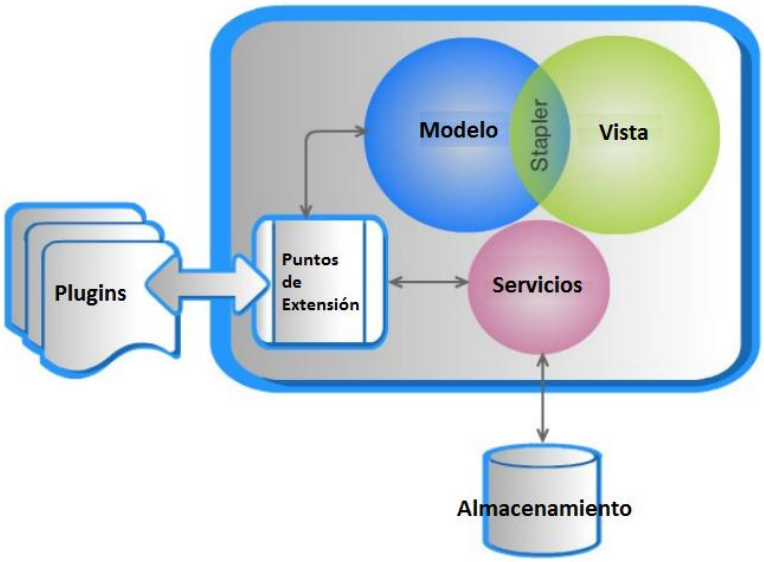


Ilustración 4 Arquitectura básica de Jenkins
Ferguson, (2011)

3.2.2.- Modelo de Objetos

El modelo de objetos de Jenkins (Ilustración 5) son bloques de construcción de la plataforma y permiten guardar la información de las tareas de un proyecto y usuarios en específico, además del estado de las ejecuciones de dichas tareas.

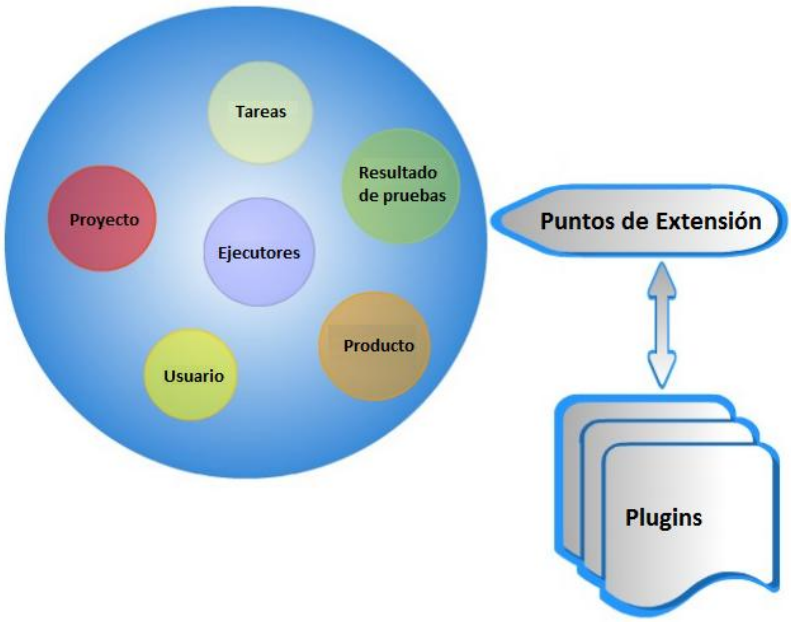


Ilustración 5 Jenkins - Modelo de Objetos
Ferguson, (2011)

Cada objeto del modelo está asociado con un URL como se muestra en la Ilustración 6, por medio de la herramienta *Stapler*[24]. *Stapler* es una biblioteca que liga o asocia los objetos de una aplicación a direcciones URL, por lo que es más fácil escribir aplicaciones web. La idea central de ésta librería es asignar automáticamente direcciones *URL* para los objetos, creando una jerarquía *URL* intuitiva.

[24] Stapler página web http://sourceforge.net/projects/stapler/

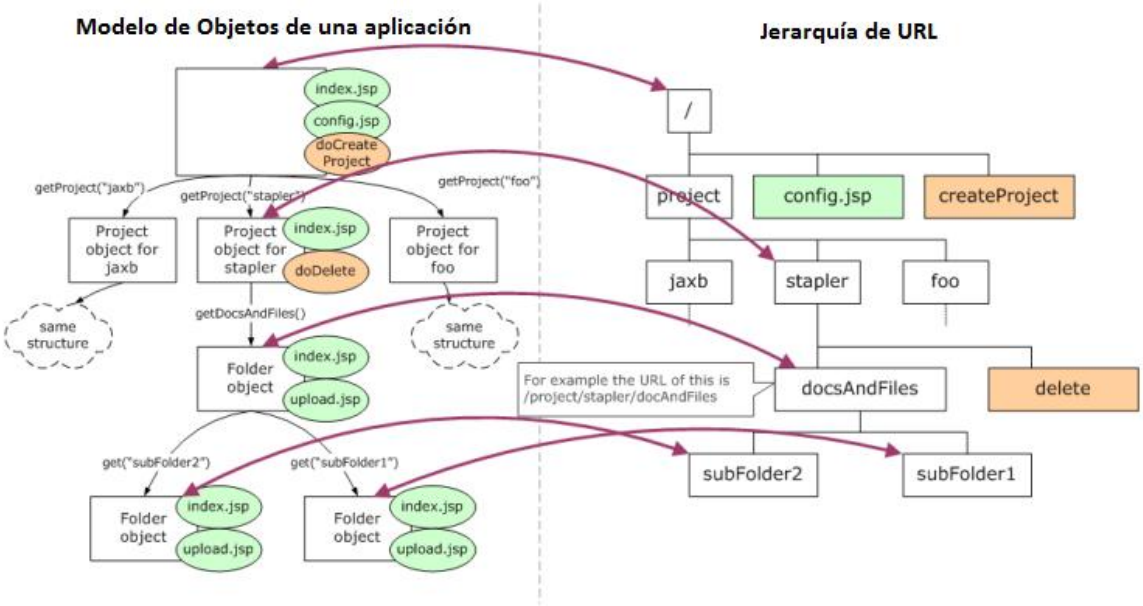


Ilustración 6 Asignación de URL a los objetos del modelo
Ferguson, (2011)

Stapler resuelve modelo de objetos muy similar a *JSF Expression Resolver*, que toma un objeto y una *URL* y posteriormente evalúa la *URL* contra el objeto. Este proceso se repite hasta que llega a un recurso estático, una vista (por ejemplo *JSP*, *Jelly*, *Groovy*, entre otras.), o a un método de acción.

3.2.3.- Modelo de Presentación

El modelo de objetos de Jenkins tiene múltiples vistas (*views*) que se utilizan para crear las páginas *HTML* de cada objeto. Jenkins utiliza la librería *Jelly*[25] como la tecnología de presentación o vista. En la Ilustración 7, se muestra el modelo de presentación de Jenkins, donde los objetos Acción/Procesable agregan la interfaz de usuario y la información al modelo de objetos; los objetos Descriptor agregan la configuración de la interfaz de usuario al modelo de objetos.

[25] https://wiki.jenkins-ci.org/display/JENKINS/Basic+guide+to+Jelly+usage+in+Jenkins

Jelly es un lenguaje de programación basado en *XML* utilizado ampliamente en los niveles más bajos de Jenkins, la ventaja que ofrece *Jelly* es que se pueden usar variables de Jenkins dentro de un script para conocer el estado de la compilación de la aplicación.

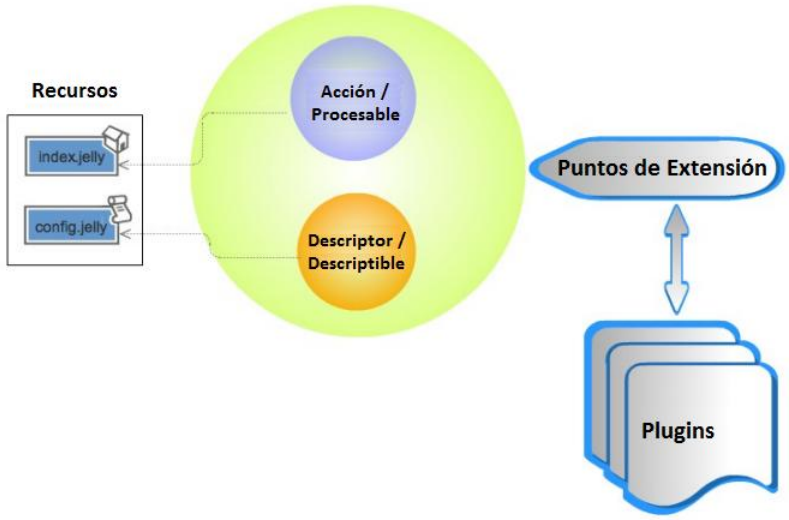


Ilustración 7 Jenkins - Modelo de Presentación
Ferguson, (2011)

Un modelo de objetos puede tener asociado vistas, que son las entradas a los motores de plantillas que se utilizan principalmente para crear archivos *HTML*. Las vistas se colocan como recursos, organizados por sus nombres de clase.

3.2.4.- Modelo de Servicios

Los objetos de servicio de Jenkins son modelos de objetos que son ejecutables. En la Ilustración 8, los objetos ejecutores (Constructor, Editor, Reporteador y Notificador) de *Jenkins* permiten ejecutar estos servicios para completar una acción en específico.

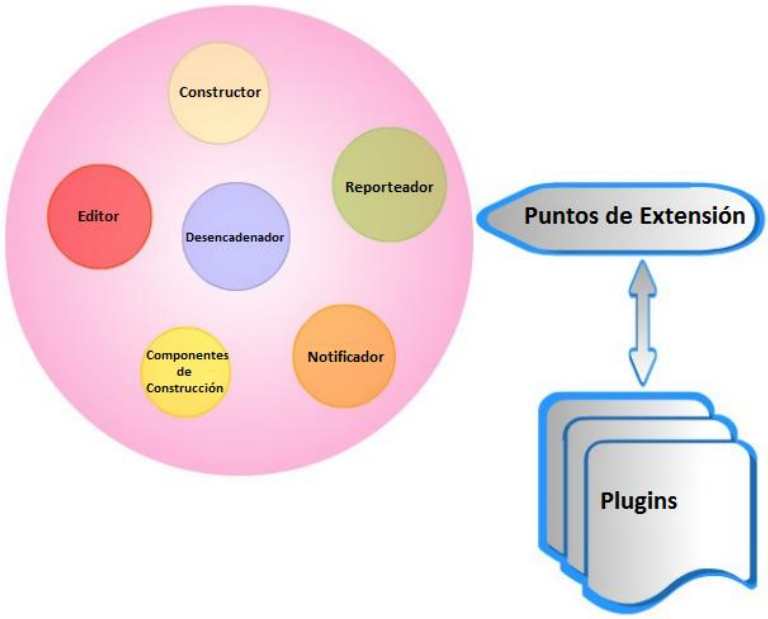


Ilustración 8 Jenkins – Servicios
Ferguson, (2011)

3.2.5.- Arquitectura de Plugins

Como ya se había mencionado antes, Jenkins es una aplicación web extendible, por medio de *plugins*, los cuales son los responsables de realizar ciertas funcionalidades de la aplicación. En la Ilustración 9, se presenta la arquitectura de *plugins*. El uso de *plugins* se logra por medio de los siguientes componentes básicos:

- Administrador de Plugins
- Centro de actualización y actualización del sitio
- Envoltorio de *Plugins* y la estrategia *Plugin*

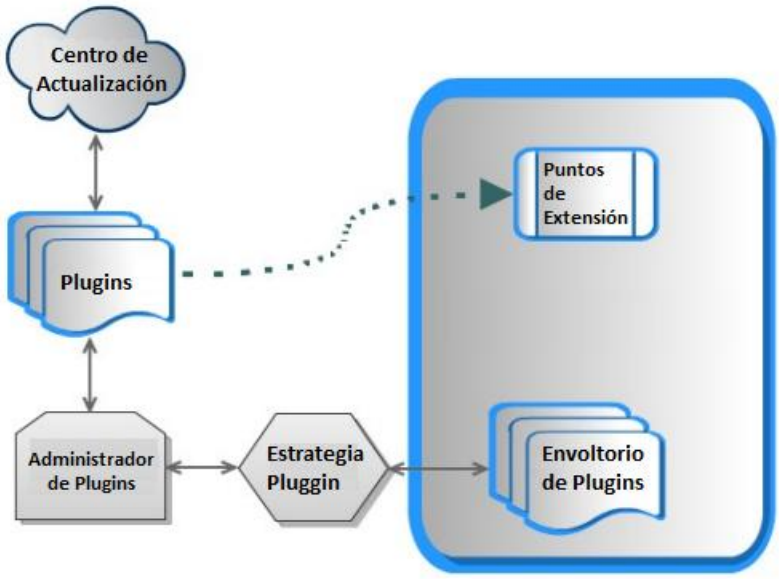


Ilustración 9 Jenkins - Arquitectura de Plugins
Ferguson, (2011)

3.2.5.1.- Administrador de Plugins

Es un servicio en el núcleo de Jenkins que es responsable de cargar los *plugins* del centro de actualizaciones, los servicios que se encuentran soportados son: la instalación de un *plugin*, cargar un *plugin* a la aplicación, configurar el centro de actualizaciones, configurar el proxy, listar los *plugins* instalados y los *plugins* disponibles. Además nos proporciona una interfaz de usuario para realizar la configuración del proxy, para la actualización de la configuración del sitio, así como las pantallas para presentar la lista de los *plugins* instalados, de los *plugins* disponibles, de las actualizaciones de *plugins* ya instalados, entre otras. En la Ilustración 10, se muestran las interfaces de usuario generadas por el *plugin* manager.

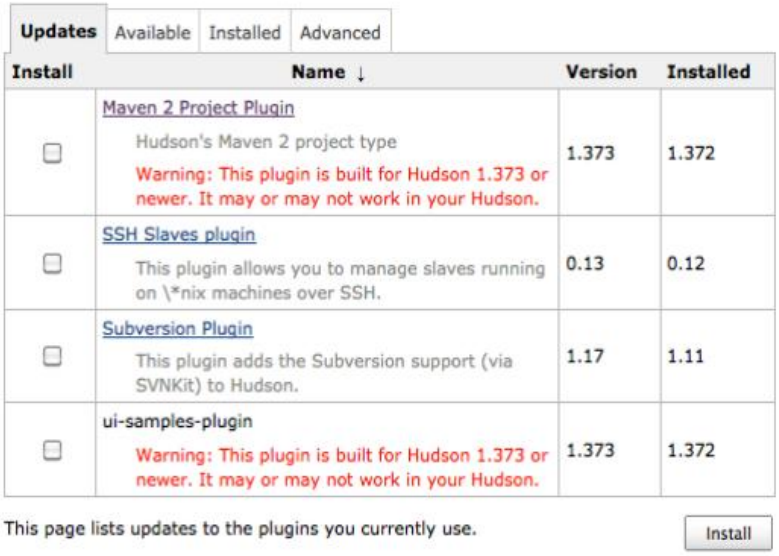


Ilustración 10 Vistas del Plugin Manager
Ferguson, (2011)

Hay un gran ecosistema de *plugins* disponibles de código abierto de terceros, lo que le permite añadir características adicionales al servidor, en tareas de compilación, de apoyo a las diferentes herramientas de SCM como *Git*, *Mercurial* o *ClearCase*, a la calidad del código y las métricas de cobertura de código, entre otras.

3.2.5.2.- Centro de Actualización y renovación del sitio

El centro de actualización y renovación del sitio, son dos modelos de objetos que tiene información sobre los sitios y *plugins* disponibles para instalar y actualizar. Jenkins soporta múltiples sitios de actualización. El administrador de *Plugins* utiliza estos objetos del modelo para proporcionarle información al administrador de *Plugins UI*. Es el encargado de ejecutar algunas tareas como realizar la conexión a un sitio de actualización, la de descarga e instalación de *plugins*, la de actualizar *plugins* ya instalados y la de realizar la actualización del núcleo de Jenkins; estas tareas se ejecutan de forma asíncrona cuando se reciben las solicitudes de la interfaz de usuario, tal como se muestra en la Ilustración 11.

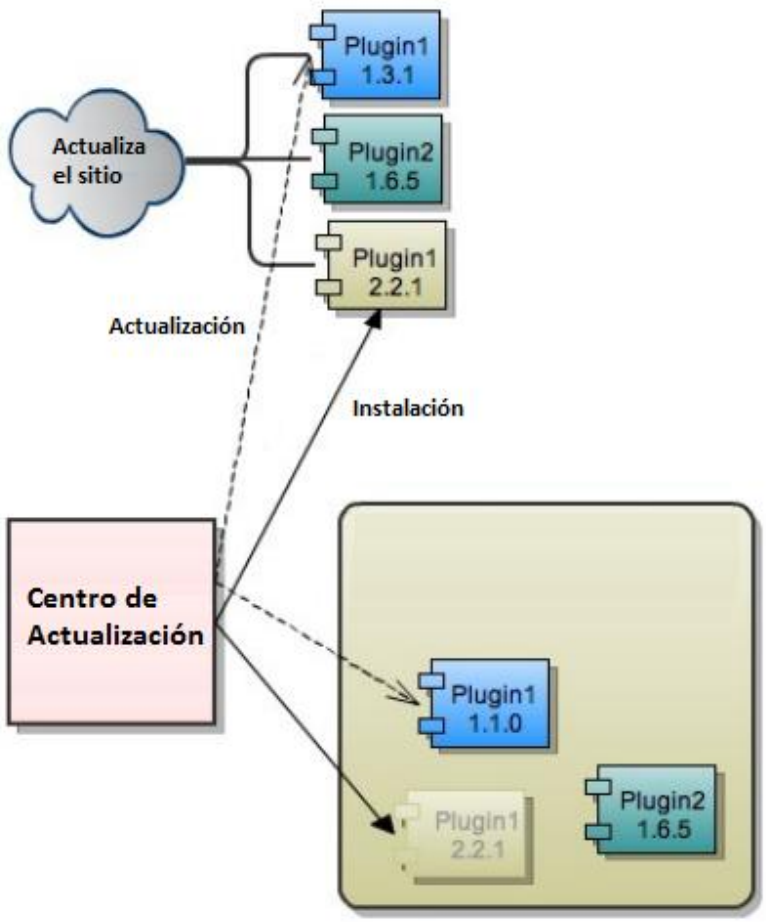


Ilustración 11 Jenkins - Update Center y Update Site
Ferguson, (2011)

3.2.5.3.- Envoltorio de Plugins y la estrategia Pluggin

Un *plugin* en *Jenkins* es un archivo del tipo *jar* con extensión *hpi*, cuyas propiedades se definen a través de un archivo *Manifest*. El punto de entrada es a través de una clase que hereda de la clase base llamada *Plugin*. Un *plugin* está destinada a un espacio en la estructura de archivos de Jenkins, los deja en la ruta <*server-root*>/*plugins*/*plugin*-nombre, donde *plugin*-nombre es tomado del nombre del *plugin* (*plugin-name.hpi*).

El envoltorio de *Plugins* es un objeto de servicio que provee la ejecución de la activación o la desactivación de un *plugin*, entre otros. Un *plugin* puede tener una página opcional llamada *config.jelly* y si está presente, se convertirá en una parte de la página de configuración del sistema.

3.3.- SonarQube

SonarQube es una herramienta de código abierto que se distribuye bajo la licencia *LGPL* v3 para gestionar la calidad del código, dicha herramienta controla la calidad en los siguientes ejes: comentarios, reglas de codificación, duplicación de código, pruebas unitarias, complejidad del código y defectos potenciales (Gigleux & Campbell 2016). *SonarQube* es una plataforma de código abierto utilizado por los equipos de desarrollo para gestionar la calidad del código fuente. *SonarQube* se ha desarrollado con un objetivo en mente: hacer que la gestión de la calidad del código accesible a todo el mundo con un mínimo de esfuerzo, por lo que *SonarQube* ofrece analizadores de código, herramientas de informes, revisiones manuales, módulos de detección de defectos y máquinas de tiempo como funcionalidades básicas. También viene con un mecanismo de *plugin* que permite a la comunidad ampliar la funcionalidad de la herramienta (Arapidis 2012).

Esta herramienta soporta más de 25 lenguajes de programación, dentro de los cuales puedo mencionar los siguientes *ABAP, C/C++, C#, COBOL, Erlang, Flex / ActionScript, Groovy, Java, JSON, PHP, PL/SQL, Python, VB, .NET, XML* entre otros por medio del uso de *plugins*.

3.3.1.- Arquitectura de SonarQube

La arquitectura de *SonarQube* mostrada en la Ilustración 12 está compuesta por 4 componentes:

Componente 1 (Servidor *SonarQube*).- es un servidor web usado por desarrolladores para navegar por las métricas de calidad de los proyectos analizados y la configuración de la herramienta *SonarQube*.

Componente 2 (Base de Datos).- usada para almacenar la información de la configuración de la instancia *SonarQube* (seguridad, *plugins* configuración, etc.) y la calidad de los proyectos, vistas, etc.

Componente 3 (*Plugins* de *SonarQube* instalados).- componentes de software que hacen extensible la aplicación.

Componente 4 (Escáneres *SonarQube*).- Estos componentes son los encargados de construir proyectos y de analizarlos.

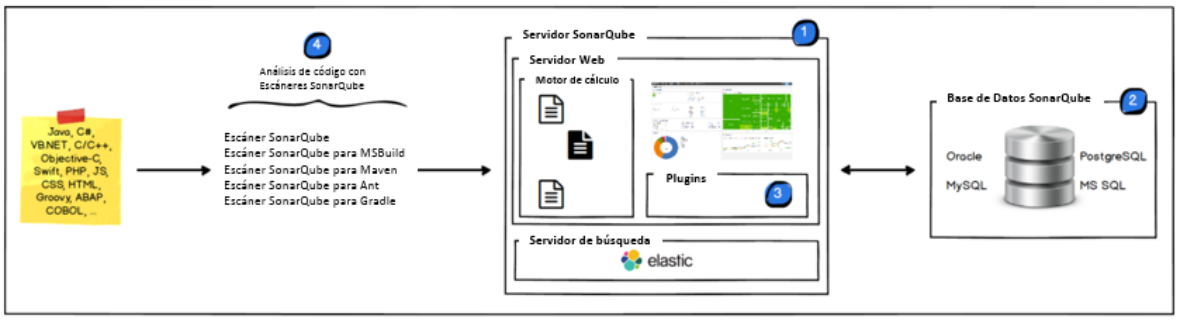


Ilustración 12 Arquitectura SonarQube
Gigleux & Campbell, (2016)

3.3.2.- Integración con SonarQube

SonarQube se puede integrar con otras herramientas *ALM (Application Lifecycle Management)* como se muestra en la Ilustración 13.

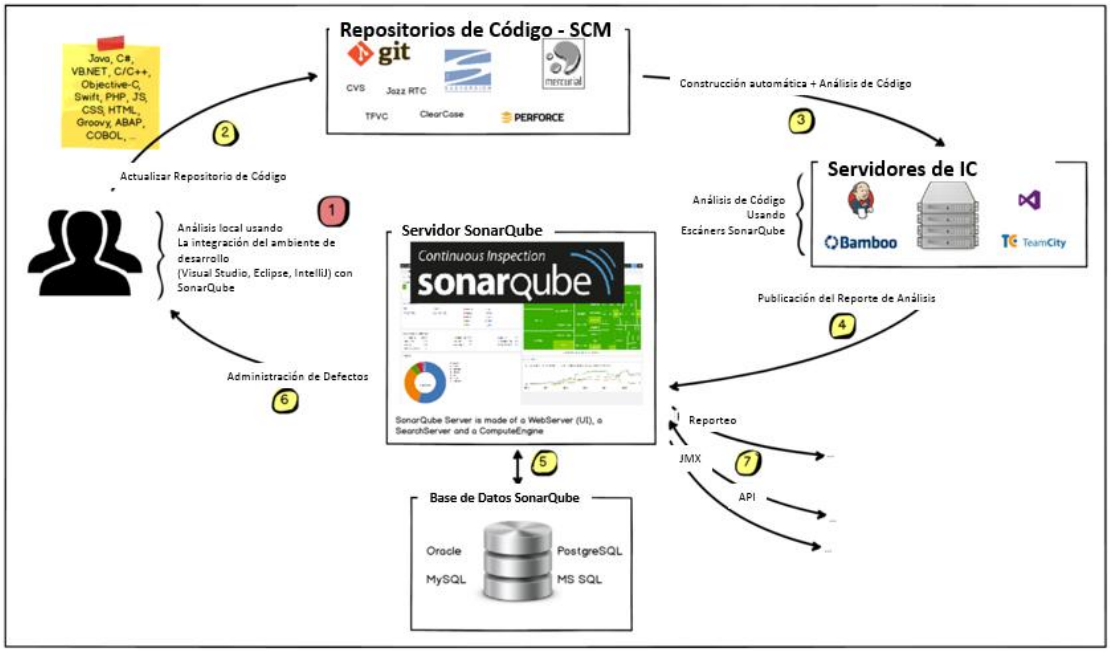


Ilustración 13 Integración con SonarQube
Gigleux & Campbell, (2016)

Se describen los componentes de la ilustración anterior:

Los desarrolladores de código en sus ambientes integrados de desarrollo (IDE) y usando la herramienta *SonarLint* para ejecutar el análisis de calidad de forma local.

Los desarrolladores suben su código en su repositorio de código (SCM) favorito.

El servidor de IC desencadena una construcción automática y la ejecución del escáner *SonarQube* requerido para ejecutar el análisis.

El informe de análisis se envía al servidor *SonarQube* para su procesamiento.

SonarQube procesa y almacena los resultados del análisis en la base de datos *SonarQube* y muestra los resultados en la interfaz de usuario.

Los desarrolladores opinan, hacen comentarios, solucionan sus defectos y reducen su deuda técnica a través de la interfaz de usuario llamada *SonarQube*.

Los gerentes e integrantes del equipo de desarrollo reciben informes de análisis realizado.

Una vez que la plataforma *SonarQube* se ha instalado, y ha instalado un analizador está listo para empezar a crear proyectos. Un proyecto se crea en la plataforma de forma automática en su primer análisis.

3.3.3.- Alcance de análisis: Tipos de archivos y datos

SonarQube puede realizar un análisis en más de 20 idiomas diferentes (Gigleux & Campbell 2016). El resultado de este análisis será medidas de calidad y los problemas (casos en los que se rompieron las reglas de codificación). Sin embargo, lo que se analiza variará en función del lenguaje de programación:

- En todos los lenguajes, los datos serán automáticamente importados de los proveedores SCM compatibles.

- En todos los lenguajes, se realiza un análisis estático de código fuente (archivos *Java*, COBOL, etc.)
- Un análisis estático del código compilado se puede realizar para determinados lenguajes (archivos *.class* de *Java*, archivos *.dll* en C#, etc.)
- Un análisis dinámico de código se puede realizar en determinados lenguajes.

Hay tres paradigmas diferentes para el análisis *SonarQube*. Puede cambiar entre los tres modos utilizando el parámetro de análisis *sonar.analysis.mode* con uno de estos tres valores:

- *publish* (publicar). - este es el valor predeterminado. Este modo analiza todo lo que es analizable para el lenguaje de programación seleccionado y envía los resultados al servidor para su procesamiento.
- *preview* (vista previa).- se utiliza normalmente para determinar si los cambios de código son lo suficientemente bueno para seguir adelante con el análisis.
- *issues* (defectos).- es un equivalente "vista previa" destinados a ser utilizados por las herramientas.

3.3.4.- Métricas obtenidas por SonarQube

Las métricas que se obtienen con la herramienta *SonarQube* son (Campbell & Papapetrou 2014), (Gigleux & Campbell 2016):

- Complejidad.- se refiere a la complejidad ciclomática (métrica para evaluar el diseño de software), y denota el número de rutas linealmente independientes del código.
- Código duplicado.- es una métrica básica de calidad y junto con la complejidad ciclomática, es el mayor enemigo de la mantenibilidad.
- Documentación.- es la métrica que muestra el porcentaje de documentación que tienen un proyecto.
- Comentarios.- indica el porcentaje de código documentado en un proyecto.
- Tamaño de código.- muestra las métricas de líneas de código, número de instrucciones, archivos, clases, paquetes, métodos entre otros.

- Defectos.- despliega el número de violaciones a las reglas de codificación de un proyecto.
- Pruebas.- Obtiene las métricas de las pruebas unitarias definidas en una aplicación.

3.3.5.- Plugins de SonarQube

Un *plugin* de *SonarQube* es un conjunto de objetos *Java* que implementan puntos de extensión. Estos puntos de extensión son interfaces o clases abstractas, que definen los contratos de lo que debe ser implementado, dichas interfaces son proporcionadas por el *API SonarQube*.

Un punto de extensión es un punto de la aplicación donde el código del *plugin* puede ser invocada. Los puntos de extensión son generalmente interfaces que pueden ser implementadas por *plugins*.

La importancia de los *plugins* de *SonarQube* radica en que permiten añadir diferentes funcionalidades, nuevas métricas, modos de visualizar datos, análisis a nuevos lenguajes de programación, integración con otras herramientas, entre otras.

Las métricas y las medidas son dos conceptos diferentes en *SonarQube*. Una métrica es una definición y/o descripción, mientras que una medida es el resultado de un cálculo realizado por el análisis (Gigleux & Campbell 2016).

3.4.- Metodología de desarrollo de software

El diseño de software no solo requiere de conocimientos y experiencia sino también de una metodología a seguir, ya que se ha demostrado que la aplicación de una metodología proporciona puntos de control y revisión, define las actividades a realizar durante el ciclo completo de desarrollo de software y unifica los criterios en el equipo de desarrollo. Una metodología define procedimientos, artefactos resultantes, funciones, herramientas y normas de calidad para el ciclo de vida del desarrollo de sistemas. Para el presente trabajo de investigación de aplicará la metodología *OpenUP*.

3.4.1.- OpenUP (Open Unified Process)

OpenUP es un proceso unificado que aplica enfoques iterativos e incrementales dentro de un ciclo de vida estructurado, utiliza la filosofía ágil que se enfoca en la naturaleza de colaboración en el desarrollo de software; es una metodología de desarrollo de software, basada en *RUP* (*Rational Unified Process*), que contiene el conjunto mínimo de prácticas que ayudan a un equipo de desarrollo de software a realizar un producto de alta calidad, de una forma eficiente: es un proceso unificado, iterativo e incremental, que se centra en el desarrollo colaborativo de software para generar sistemas de calidad, como se muestra en la Ilustración 14; es un proceso modelo y extensible, dirigido a la gestión y desarrollo de proyectos de software basados en un desarrollo iterativo, ágil e incremental apropiado para proyectos pequeños y de bajos recursos, es aplicable a un conjunto amplio de plataformas y aplicaciones de desarrollo. Sin embargo *OpenUP* es completa en el sentido de que manifiesta por completo el proceso de construir un sistema y para atender las necesidades que no están cubiertas en su contenido *OpenUP* es extensible a ser utilizado como base sobre la cual se pueden añadir o adaptar al contenido de otra metodología (Eclipse Foundation 2012).

Ilustración 14 Principios básicos de OpenUP
Eclipse Foundation, (2012)

OpenUP divide el contenido metodológico del contenido procedimental. El contenido metodológico es el que define elementos metodológicos tales como disciplinas, tareas, artefactos y procesos, independientemente de la forma en que se usen estos o se combinen, como se presenta en la Ilustración 15.

Ilustración 15 Contenido metodológico de OpenUP
Eclipse Foundation, (2012)

Por el contrario, el contenido procedimental, es donde se aplican todos estos elementos metodológicos dentro de una dimensión temporal, pudiéndose crear multitud de ciclos de vida diferentes a partir del mismo subconjunto de elementos metodológicos. Todo proyecto en *OpenUP* consta de cuatro fases (Ilustración 16): inicio, elaboración, construcción y transición (heredada de *RUP*), donde cada fase se divide a su vez en iteraciones.

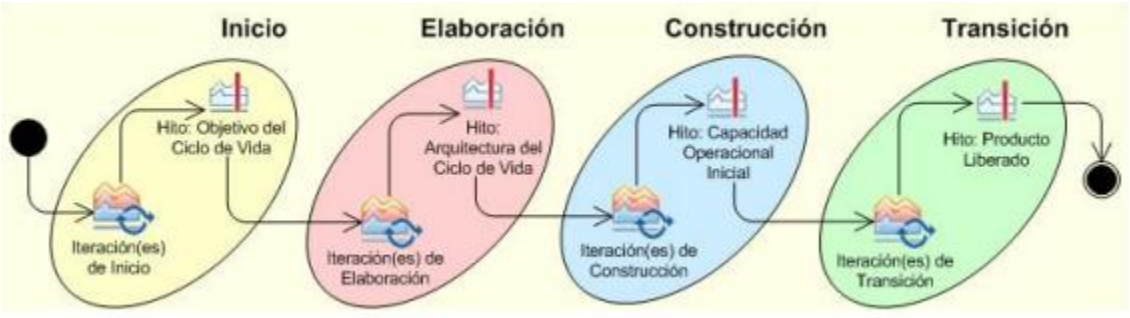


Ilustración 16 Fases de OpenUP
Eclipse Foundation, (2012)

3.4.2.- Fases de OpenUP

3.4.2.1.- Inicio

En esta fase, las necesidades son tomadas en cuenta y plasmadas en objetivos del proyecto. Se definen para el proyecto: el ámbito, los límites, los criterios de aceptación, los casos de uso críticos, una estimación inicial del costo y una planificación inicial.

Objetivos:

- Entender qué construir.
- Identificar la funcionalidad principal o clave.
- Determinar al menos una posible solución.
- Entender costos, calendario y riesgos del proyecto.

3.4.2.2.- Elaboración

En esta fase se llevan a cabo las tareas de análisis del dominio y la definición de la arquitectura del sistema. Se debe elaborar un plan de proyecto, estableciendo unos requisitos y una arquitectura estables. También se especifica el proceso de desarrollo, las herramientas, la infraestructura a utilizar y el entorno de desarrollo. Al final de esta fase se debe tener una definición clara y precisa de los casos de uso, los actores, la arquitectura del sistema y un prototipo ejecutable del sistema.

Objetivos:

- Obtener un entendimiento con mayor nivel de detalle de los requerimientos.
- Diseñar, implementar y validar la línea base arquitectónica.
- Mitigar riesgos y lograr estimaciones de costos y calendarios más precisos.

3.4.2.3.- Construcción

En esta fase todos los componentes y funcionalidades del sistema que falten por implementar son realizados, probados e integrados. Los resultados obtenidos en forma de incrementos ejecutables deben ser desarrollados de la forma más rápida posible sin dejar de lado la calidad de lo desarrollado.

Objetivos:

- En forma iterativa, desarrollar un producto para que sea probado.
- Minimizar los costos de desarrollo y lograr cierto nivel de paralelismo.

3.4.2.4.- Transición

Esta fase corresponde a la introducción del producto en la comunidad de usuarios, cuando el producto está lo suficientemente maduro. La fase de la transición consta de las subfases de pruebas de versiones beta y capacitación de los usuarios finales y de los encargados del mantenimiento del sistema. En función de la respuesta obtenida por los usuarios puede ser necesario realizar cambios en las entregas finales o implementar alguna funcionalidad más.

Objetivos:

- Realizar las pruebas beta para determinar si se alcanzaron las expectativas de los usuarios.
- Alcanzar la concordancia con las partes interesadas de que el producto está terminado.
- Mejorar el rendimiento (performance) de la aplicación por medio del análisis retrospectivo del proyecto.

3.4.3.- Entregables

OpenUP define entregables por fase, dichos artefactos se especifican en la Tabla 5.

Fase	Actividad	Artefacto
Inicio	Inicio del proyecto	Documento de visión
	Planeación del proyecto	Plan de desarrollo del software
	Identificación de requerimientos	Especificación de requisitos de software
Elaboración	Desarrollo de la arquitectura	Documento de la arquitectura del sistema
Construcción	Documento de Iteración	Plan por Iteración
	Diseño detallado	Documento de diseño detallado
	Casos de Uso	Especificación de casos de uso
	Definición de pruebas de la aplicación	Plan de pruebas del software
	Creación de casos de prueba	Especificación de casos de prueba
	Construcción de la aplicación	Código Fuente Ejecución de pruebas unitarias
Transición	Despliegue de la aplicación	Manual de instalación Documentación técnica

Tabla 5 Entregables de OpenUP
Elaborada por Romero Ricardo, (2016)

Se escogió esta metodología porque es apropiada para proyectos pequeños y de bajos recursos, permite disminuir las probabilidades de fracaso e incrementar las probabilidades de éxito; permite detectar errores tempranos a través de ciclos iterativos; evita la elaboración de documentación, diagramas e iteraciones innecesarios requeridos en la metodología *RUP*; por ser una metodología ágil tiene un enfoque centrado al cliente, con iteraciones cortas y porque se pueden añadir o adaptar al contenido de otra metodología.

4.- DESARROLLO

4.1.- Fase Inicio

Como la primera de las cuatro fases del ciclo de vida de un proyecto de software, la fase de inicio trata de comprender el alcance y los objetivos del proyecto y obtener suficiente información para confirmar la viabilidad del proyecto.

Los proyectos pueden tener una o más iteraciones en la fase inicial, ya que si el proyecto es muy grande, con una sola iteración es difícil de definir su alcance, definir sus relaciones y detectar los riesgos técnicos importantes.

4.1.1 Visión del proyecto

En el APÉNDICE 1, se presenta el documento Visión del proyecto, artefacto donde se plasma una base de alto nivel para los requerimientos técnicos del proyecto; captura la solución técnica mediante la descripción de las peticiones de las partes interesadas y las limitaciones que dan una visión general del razonamiento, antecedentes y contexto para posteriormente analizar los requerimientos de forma más detallada; la visión sirve para la comunicación del proyecto y proporciona una estrategia con la cual todas las decisiones futuras puedan ser validadas; debe reunir a los miembros del equipo detrás de una idea y darles el contexto para la toma de decisiones en el área de requerimientos, esta visión debe estar disponible para todos los participantes del equipo.

Como una buena práctica, este documento se debe de liberar a los interesados tan pronto como sea posible, debe de ser redactado de forma tal, que a las partes interesadas les sea fácil leer y entender. Debe de incluir sólo las características más importantes y evitar detallar los requerimientos.

En éste artefacto se detalla el problema que se va a resolver, cabe aclarar que el problema afecta a arquitectos de software e ingenieros de software y el impacto que tiene es que la solución puede ser usado como referencia en la toma de decisiones para la reducción de costos y para la mitigación de riesgos en empresas de desarrollo de software; se presenta

tanto la descripción como las responsabilidades de arquitectos de software e ingenieros de software; se muestran las necesidades, sus prioridades y la versión en que se planea cubrir dicha necesidad, así como otros requerimientos que dicha solución debe de cubrir.

4.1.2 Plan del proyecto

En este mismo apéndice, se muestra el artefacto llamado Plan del proyecto. Este artefacto reúne toda la información necesaria para gestionar el proyecto a nivel estratégico. Su parte principal consiste en un plan general, para obtener la identificación de iteraciones del proyecto y sus objetivos específicos. El propósito de este artefacto es proporcionar un documento central donde cualquier miembro del equipo del proyecto puede encontrar la información sobre cómo se gestionará el proyecto.

Como una buena práctica, este artefacto debe de ser actualizado con la frecuencia necesaria, por lo general al final de cada iteración, con el fin de reflejar los cambios en las prioridades y necesidades, así como registro de las lecciones aprendidas del proyecto, la actualización de este artefacto se debe de realizar en reuniones de planificación que implican a todo el equipo y a las partes interesadas del proyecto, con el fin de asegurarse de que todo el mundo está de acuerdo con él.

De éste artefacto podemos citar entre las secciones más importantes las siguientes: la definición del objetivo general del proyecto, así como de los objetivos específicos; la definición de las fases, el número de iteraciones por cada fase y su duración; se especifican los hitos de cada fase, así como los objetivos a cubrir en cada iteración.

4.1.3 Requerimientos del proyecto

El último artefacto en el APÉNDICE 1, es el documento Requerimientos del proyecto. Este artefacto detalla los atributos de calidad del proyecto, así como las limitaciones que las opciones de diseño deben satisfacer para cubrir los objetivos de negocio y objetivos técnicos, captura los requerimientos funcionales que no se expresan como casos de uso, permite evaluar el tamaño, costo, y la viabilidad del proyecto propuesto y por último, permite entender los requisitos a nivel de servicio para la gestión operativa del proyecto.

Como una buena práctica, este artefacto debe documentar los requerimientos de todo el sistema, asegurándose que las necesidades de todas las partes interesadas estén representadas; debe de incluir las necesidades de los que son responsables de realizar el mantenimiento y soporte del sistema después de la entrega; los requerimientos pueden ser considerados de forma cualitativa (especificando una cualidad o característica deseable) y de forma cuantitativa (especificando la cantidad), ésta última se debe comprobar, ya sea a través de pruebas o alguna otra evaluación objetiva.

En éste artefacto se detallan los diversos requerimientos no funcionales a cubrir por el proyecto, agrupados en atributos de calidad, dichos atributos son usabilidad, confiabilidad y compatibilidad.

4.2.- Fase Elaboración

En la fase de elaboración se realiza el análisis del dominio y la definición de la arquitectura del sistema; se especifica el proceso de desarrollo, las herramientas, la infraestructura a utilizar y el entorno de desarrollo.

4.2.1 Arquitectura del sistema

En el APÉNDICE 2, se presenta el documento Arquitectura del sistema, en este artefacto se describe la arquitectura de software; proporciona un lugar para el mantenimiento de la lista de cuestiones arquitectónicas, junto con las decisiones arquitectónicas asociadas, diseños, modelos, código documentados, todo en los niveles apropiados para que sea fácil de entender qué decisiones arquitectónicas se han realizado; es útil para los arquitectos utilizar este artefacto para colaborar con otros miembros del equipo en el desarrollo de la arquitectura y para ayudar a los miembros del equipo para que entiendan la motivación detrás de las decisiones arquitectónicas.

Algunas secciones importantes de éste artefacto son: los objetivos de la arquitectura, requerimientos arquitectónicos significativos, las decisiones, limitaciones y justificaciones y las vistas arquitectónicas.

Entre los objetivos de la arquitectura para el proyecto, se pueden citar los siguientes:

- Crear el *plugin* para el servidor de integración continua Jenkins;
- Obtener el código fuente a analizar de una herramienta de control de versiones;
- Presentar el resultado de las métricas obtenidas en una página web;
- Exportar el resultado de las métricas en 3 formatos definidos (*CSV, PDF y XLS*) y
- Guardar el resultado de las métricas obtenidos en la base de datos *MySQL*.

Los requerimientos arquitectónicos significativos del proyecto está el garantizar la disponibilidad, integridad y consistencia de la información y soportar 1000 usuarios de forma concurrente.

Entre las decisiones, limitaciones y justificaciones, se pueden mencionar:

- El sistema debe permitir la integración con el Sistema de Control de Versiones configurado para obtener la información de versionamiento del activo de software a analizar;
- Debe permitir el manejo de altos volúmenes de información, facilitar exportar la información de los análisis generados en diversos formatos;
- La solución debe estar orientada a ser una plataforma colaborativa; debe ser una aplicación web, utilizando tecnologías de código abierto y como lenguaje de programación es *Java*.

La arquitectura del sistema denota el conjunto de estructuras o perspectivas de alto nivel de un sistema de software. Estas están conformadas por elementos de software, las relaciones entre estos y las propiedades tanto de dichos elementos como de sus relaciones, por lo que se presentan la vista lógica, la vista operacional y el diagrama de casos de uso.

La vista lógica (Ilustración 17) Representa las capas del sistema distribuido, los componentes de servicio del sistema son descritos en niveles de servicio de infraestructura para proporcionar asistencia a los componentes de aplicaciones de todas las capas lógicas.

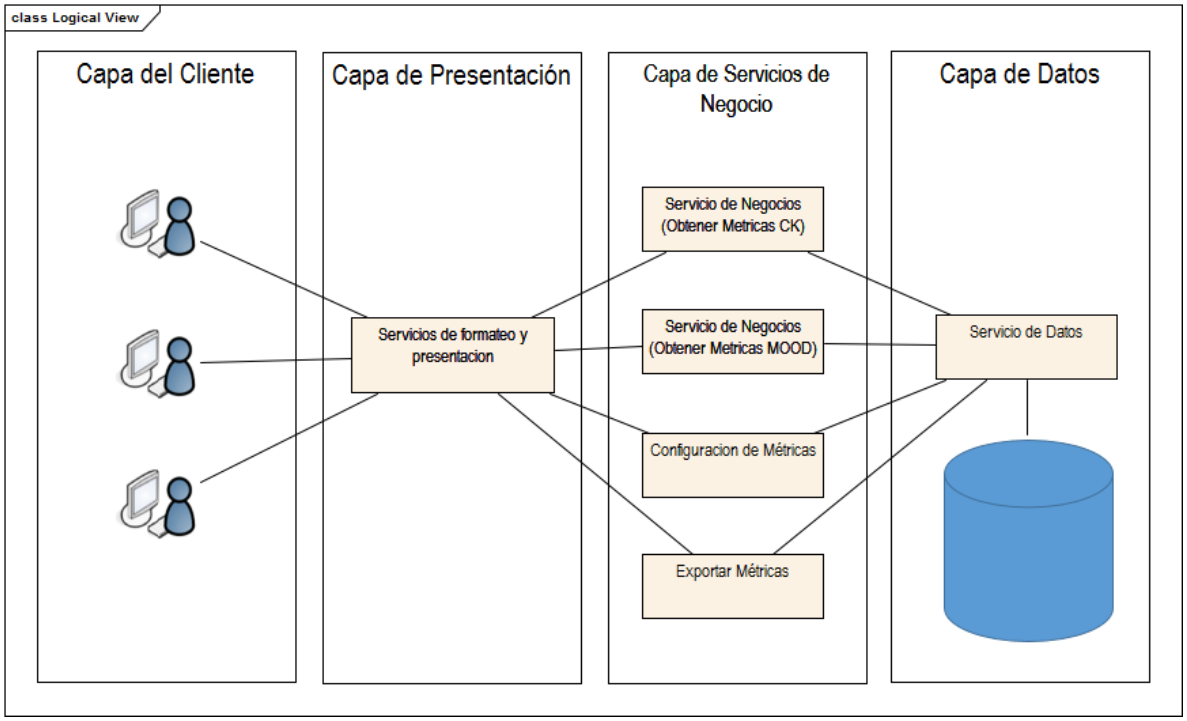


Ilustración 17 Vista Lógica
Elaborada por Romero Ricardo, (2016)

A continuación se describen las capas de la vista lógica.

- Capa de cliente. La capa de cliente está formada por la lógica de la aplicación a la que el usuario final accede directamente mediante una interfaz de usuario. La lógica de la capa de cliente podría incluir clientes basados en navegadores web que sean ejecutados en un equipo de escritorio o portátil.
- Capa de presentación. La capa de presentación está formada por la lógica de aplicación, que prepara datos para su envío a la capa de cliente y procesa solicitudes desde la capa de cliente para su envío a la lógica de negocios del servidor, por medio de los servicios de formateo y presentación.
- Capa de servicios de negocios. La capa de servicios de negocio consiste en la lógica que realiza las 4 funciones principales de la aplicación: obtener métricas CK, obtener métricas MOOD, configuración de métricas y exportación de las métricas obtenidas. El servicio de negocio "obtener métricas CK" está dedicado a analizar el código para obtener las métricas *WMC (Weighted Methods Per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling Between Object Classes), RFC*

(Response for Class), LCOM (Lack of Cohesion of Methods), MPC (Message Passing Coupling), DAC (Data Abstraction Coupling) y SIZE2; el servicio "obtener métricas MOOD" se avoca a analizar el código para obtener las métricas *PF (Polymorphism Factor), CF (Coupling Factor), MHF (Method Hiding Factor), AHF (Attribute Hiding Factor), MIF (Method Inheritance Factor) y AIF (Attribute Inheritance Factor)*; el servicio "configuración de métricas" está enfocado a la validación y actualización de los valores por defecto de las métricas y por último, el servicio "exportación de las métricas" se encuentra dedicado a la recuperación de las métricas obtenidas y a la exportación de dichas métricas en los formatos ya establecidos.

- Capa de datos. La capa de datos está formada por los servicios que proporcionan los datos persistentes utilizados por la lógica de negocios y que son almacenados en un sistema de administración de bases de datos. Básicamente ésta capa realiza el almacenamiento de las métricas obtenidas en los proyectos *Java* analizados.

Es importante señalar que la vista lógica destaca la independencia lógica y física de los componentes, representada en 4 capas separadas. Estas capas representan la separación de la lógica de la aplicación en varios equipos en un entorno de red:

Independencia lógica. Las cuatro capas del modelo arquitectónico representan independencia lógica ya que se puede modificar la lógica de la aplicación en una capa independientemente de la lógica de las otras capas. Puede cambiar la implementación de la lógica de negocios sin tener que cambiar o actualizar la lógica de la capa de presentación o la de cliente.

Independencia física. Las cuatro capas también representan independencia física porque es posible implementar la lógica en capas distintas en varias plataformas de hardware. Esta independencia permite ejecutar componentes de aplicación distribuida en los equipos que mejor se adapten a las necesidades informáticas individuales y a maximizar el ancho de banda de red.

La vista operacional (Ilustración 18) muestra los nodos físicos del sistema, las capas, las tecnologías asociadas y los componentes que se ejecutan en cada nodo.

En el diagrama operacional se muestran 5 nodos:

- El nodo "Computadora Personal" es la que contiene el navegador web en la computadora del cliente y es donde se muestra información de la aplicación al cliente. Se comunica con el nodo "Servidor de Web" mediante el protocolo *HTTP (Hypertext Transfer Protocol).*
- En el nodo "Servidor Web", reside la parte estática de la aplicación y se comunica con el nodo "Servidor de Aplicaciones" por medio del protocolo *HTTP*.
- En el nodo "Servidor de Aplicaciones", se publica y se configura la herramienta de IC (Jenkins), además de ser el nodo en donde se ejecutan las tareas para realizar el análisis de calidad del código. Se comunica con el nodo "Servidor Métricas CKMOOD" por medio del protocolo *HTTP*.
- El nodo "Servidor Métricas CKMOOD", es el que contendrá la aplicación a desarrollar y es la encargada de recolectarla información para la generación de las métricas CK y MOOD. Internamente contiene 3 capas: la capa de presentación, la capa de negocio y la capa de datos; la comunicación en éste nodo es mediante interfaces y únicamente la capa de datos es la que se comunica con el nodo "Servidor de Bases de Datos" por medio de la interface *JDBC (Java Database Connectivity).*
- Finalmente, el nodo "Servidor de Bases de Datos", contiene la herramienta de base de datos utilizada (*MySQL*) y es donde se almacena el resultado tanto del análisis realizado, como de las métricas obtenidas.

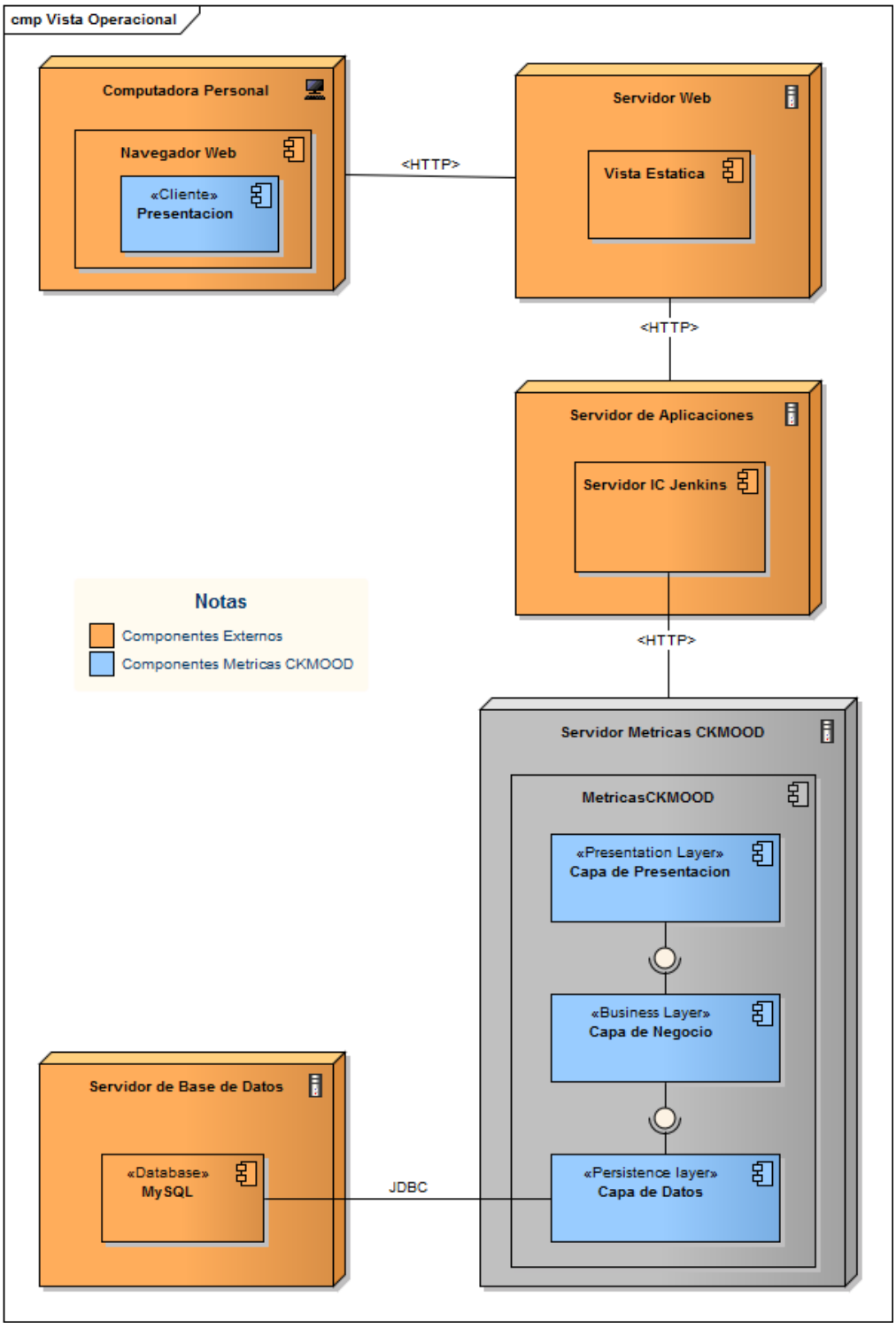


Ilustración 18 Vista Operacional
Elaborada por Romero Ricardo, (2016)

El diagrama de casos de uso (Ilustración 19) contiene las interacciones entre el usuario y el sistema, además de especificar la interacción, comunicación y relación entre los elementos del modelo que cubren los requerimientos del sistema. En dicho diagrama se presentan 2 actores que interactúan con el sistema: usuario final y un actor interno llamado Jenkins.

El actor "usuario final", va a poder modificar los valores por default de las métricas CK y MOOD por medio del caso de uso "Configurar Métricas"; y por medio del caso de uso "Exportar Métricas" va a poder descargar el resultado de las métricas en los formatos ya establecidos. Para los casos de uso "Obtener Métricas", "Obtener Métricas MOOD" y "Obtener Métricas CK", el actor que lanza dichos procesos es un actor interno "Jenkins", ya que dichos casos de uso serán ejecutados como resultado de una tarea de análisis dentro de la herramienta de IC.

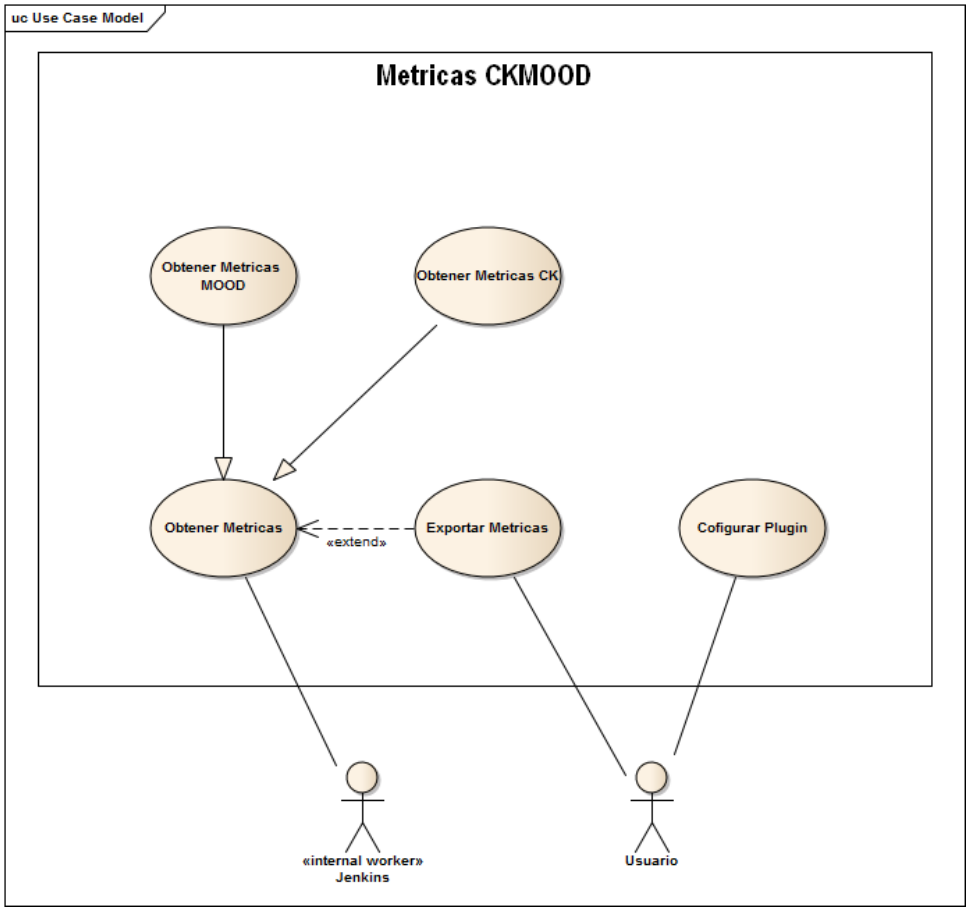


Ilustración 19 Diagrama de Casos de Uso
Elaborada por Romero Ricardo, (2016)

4.3.- Fase Construcción

En la fase de construcción se realiza la implementación de la solución propuesta, además del diseño y la ejecución de las pruebas de la solución desarrollada. En el APÉNDICE 3, se presentan los artefactos que forman parte de ésta fase, mismos que a continuación se detallarán.

4.3.1 Plan de Iteración

El primer artefacto de la fase de construcción, es el llamado Plan de Iteración, éste artefacto define los objetivos a cubrir en cada iteración planeada del proyecto; la planeación de la asignación del trabajo para cubrir con dichos objetivos y el responsable del mismo; los criterios de evaluación de la iteración; la evaluación de las pruebas realizada en relación con los objetivos planteados; y por último se pueden registrar las preocupaciones o desviaciones detectadas, en relación con los objetivos, que son importantes para planear otra fase de corrección de defectos.

Como la fase de construcción se dividió en 2 iteraciones, se generaron dos artefactos. En el primer artefacto llamado Plan por Iteración 1 los objetivos de alto nivel a cubrir son

a. análisis de librerías para la generación de métricas de la clasificación de código abierto existentes en el mercado
b. diseñar y construir el núcleo del sistema que recolecte la información del código fuente, para generar las métricas CK y MOOD
c. crear el *plugin* para el servidor de integración continua
d. presentar el resultado de las métricas obtenidas en pantalla

También en dicho artefacto, se realiza la asignación de trabajo y su estimación de tamaño, que para la iteración 1, corresponden los primeros tres casos de uso: CU1 - Obtener métricas, CU2 – Obtener métricas CK y CU3 – Obtener métricas MOOD. En cada caso de uso, la estimación del tamaño se realiza en base a la estimación denominada “Puntos Casos de Uso” (*Use Case Points* por sus siglas en inglés), la cual categoriza el tamaño de un caso de uso, en tres posibles opciones:

- Simple: Un caso de uso simple, es cuando la interfaz de usuario es muy sencilla; interactúa con una sola entidad de base de datos; tiene escenarios con 3 pasos o menos; su implementación implica menos de 5 clases.
- Promedio: Un caso de uso promedio, es cuando la interfaz de usuario tiene más diseño; interactúa con 2 o más entidades de base de datos; tiene escenarios que van desde 4 hasta 7 pasos; su implementación implica entre 5 y 10 clases.
- Complejo: Un caso de uso complejo, es cuando la interfaz de usuario es muy compleja o es un procesamiento de datos; interactúa con 3 o más entidades de base de datos.; tiene escenarios con más de 7 pasos; su implementación involucra a más de 10 clases.

Los criterios de evaluación para la iteración 1, también son definidos en dicho artefacto, siendo éstos:

1. construcción del 100% de la funcionalidad
2. más del 90% de casos de pruebas exitosos
3. obtención de las métricas CK y MOOD de forma correcta
4. presentación de las métricas obtenidas en una página web
5. realizar la integración correcta con el servidor de IC

En el artefacto llamado Plan por Iteración 2 los objetivos de alto nivel a cubrir son:

a. construir la funcionalidad necesaria para exportar las métricas obtenidas en los formatos antes especificados
b. construir la funcionalidad para que el usuario configure los valores de referencia de las métricas
c. realizar el refinamiento de la parte visual y de la usabilidad de la aplicación

La asignación de trabajo para la iteración 2, corresponden los últimos casos de uso: CU4 – Exportar métricas y CU5 – Configurar métricas. La estimación del tamaño de estos casos de uso, se realiza también con la estimación “Puntos Casos de Uso”.

Los criterios de evaluación para la iteración 2, definidos en el artefacto son:

1. construcción del 100% de la funcionalidad
2. más del 90% de casos de pruebas exitosos
3. exportación de las métricas obtenidas en los formatos CSV, PDF y XLS

4.3.2 Diseño detallado

El segundo artefacto, es el llamado diseño detallado y es generado para cada caso de uso. Este artefacto tiene el propósito de documentar los subsistemas y patrones de diseño usados en la implementación de la solución; también se explican las realizaciones de cada caso de uso, mostrando por una parte el diagrama de clases en donde se muestra la implementación de los patrones usados (vista de los participantes), así como el diagrama de secuencia básico, el cual muestra la interacción de los diversos componentes para obtener un objetivo en particular, que es la solución de cada caso de uso.

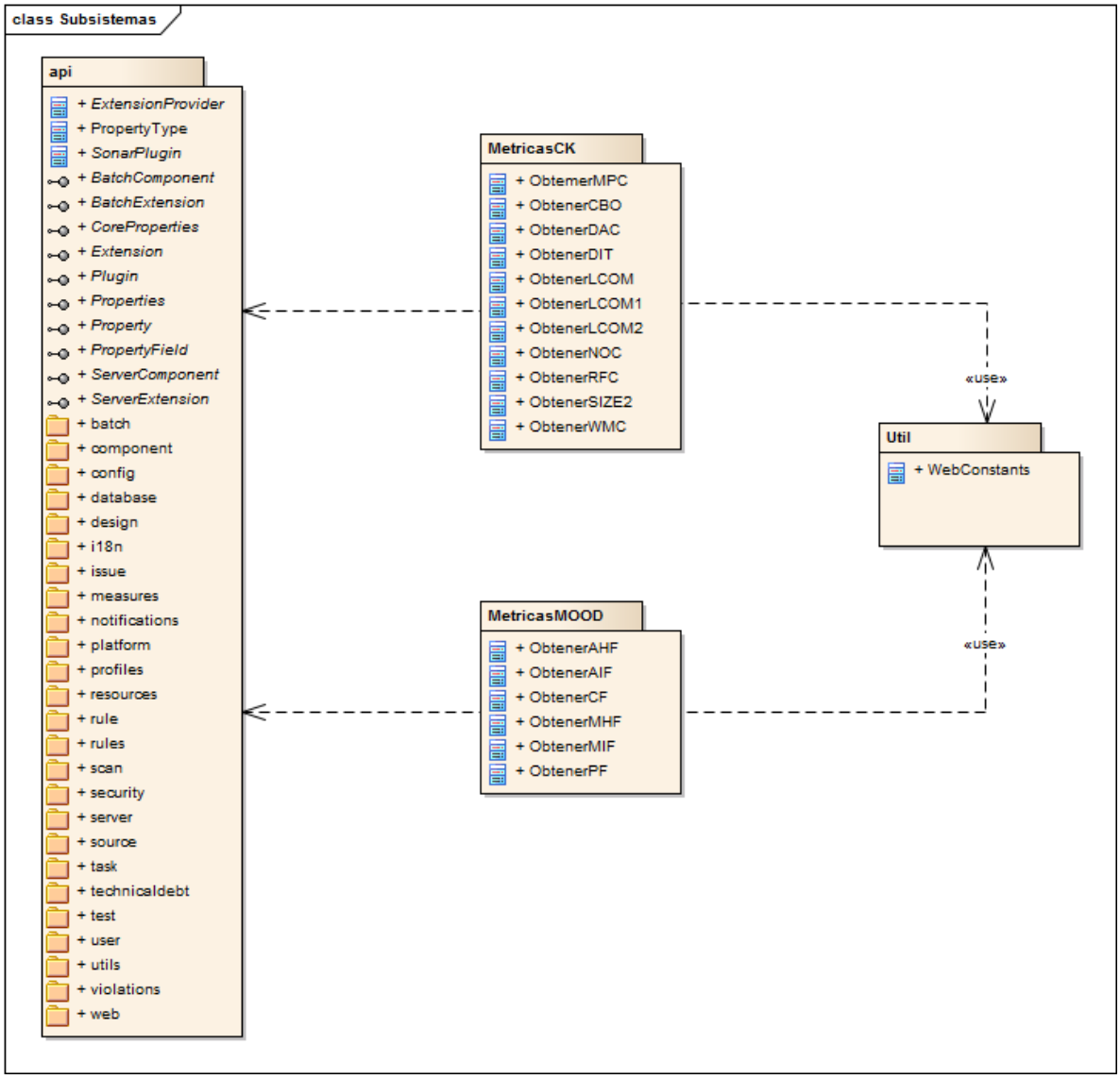


Ilustración 20 Estructura de Diseño del Plugin
Elaborada por Romero Ricardo, (2016)

El diseño de los componentes para la creación del *plugin* (Ilustración 20), observamos la api de *SonarQube*, con la cual interactúan las clases tanto de las métricas CK como MOOD, cada una de las clases que se encuentran en los paquetes llamados MétricasCK y MétricasMOOD siguen el patrón de diseño *Decorator* (Ilustración 21), el cual tiene como utilidad principal la de agregar funcionalidad de forma dinámica a objetos para ofrecer más funcionalidad de la que se tenía al principio, esto permite no tener que crear sucesivas clases que hereden de la primera para agregar nueva funcionalidad, esto es muy útil para evitar el uso de jerarquías de clases muy complejas.

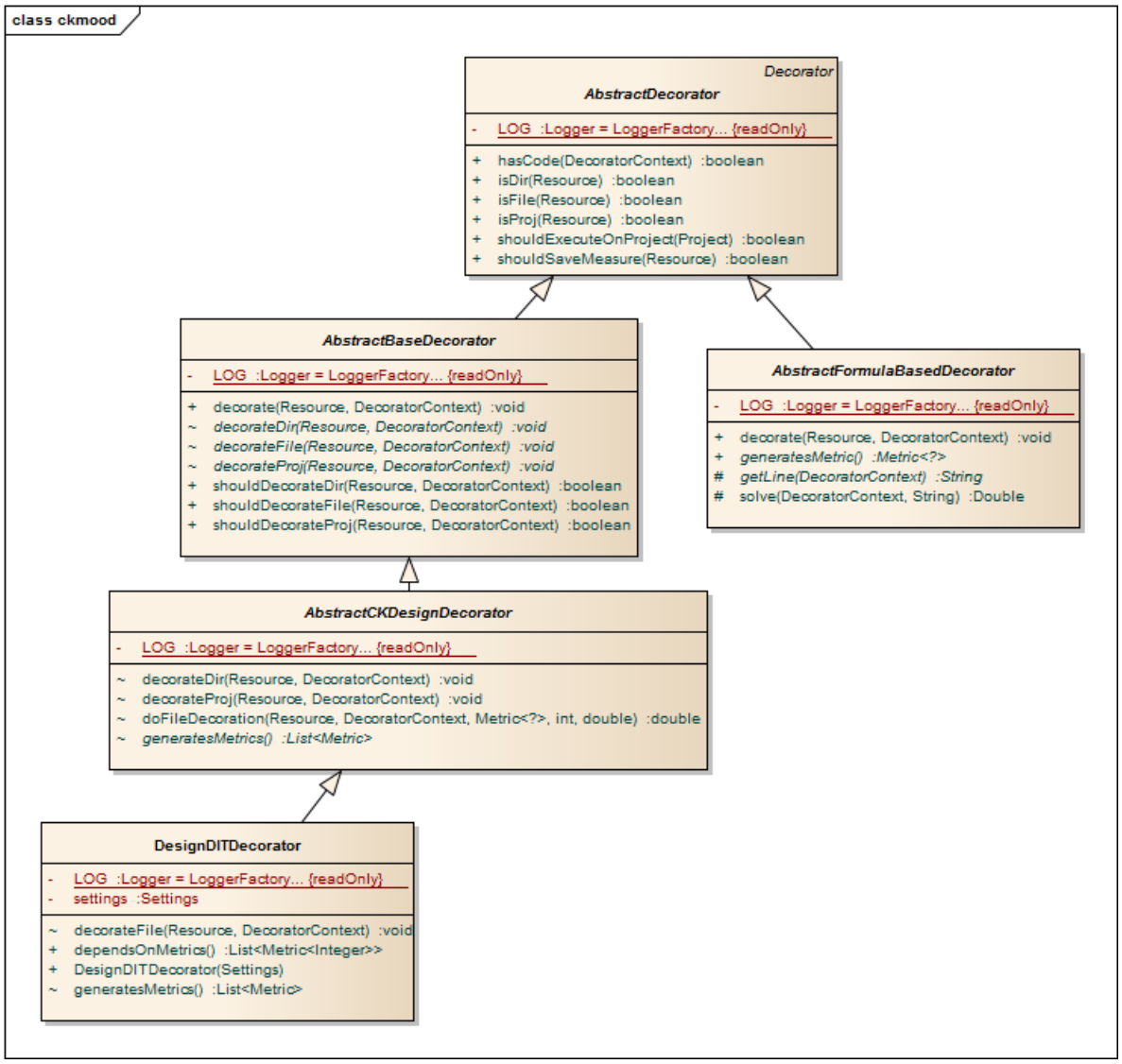


Ilustración 21 Diagrama de Clases de la Implementación del Patrón Decorator
Elaborada por Romero Ricardo, (2016)

En el diagrama de clases del patrón de diseño *Decorator* mostrado en la Ilustración 22, podemos observar los componentes participantes en la implementación del patrón *Decorator* donde: la clase ***Componente*** define la interfaz para los objetos que pueden tener responsabilidades añadidas; la clase ***ComponenteConcreto*** define el objeto al cual se le pueden agregar responsabilidades adicionales; la clase ***Decorador*** mantiene una referencia al componente asociado e implementa la interfaz de la superclase ***Componente***; las clases ***DecoradorConcretoA y DecoradorConcretoB*** añaden nuevas responsabilidades al componente ***ComponenteConcreto***.

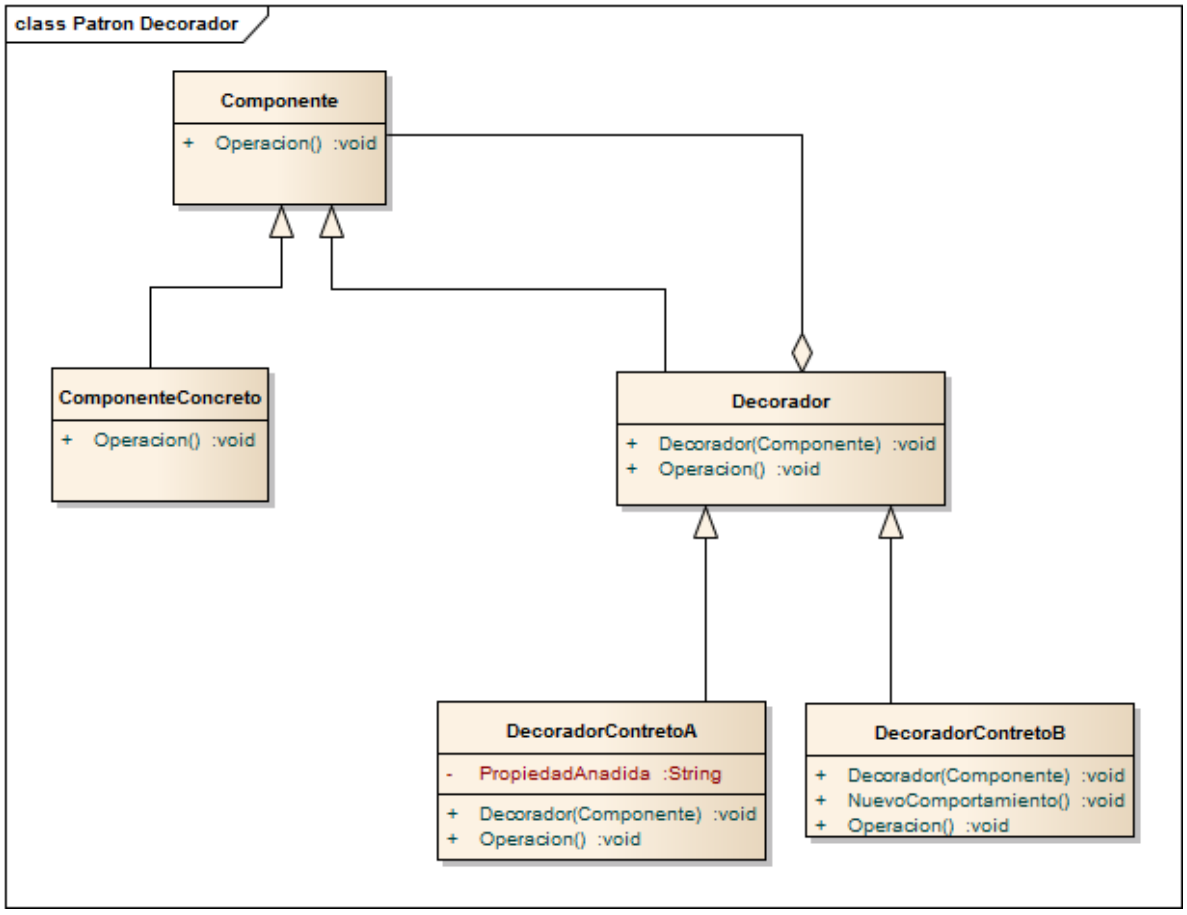


Ilustración 22 Diagrama de Clases del Patrón Decorator
Elaborada por Romero Ricardo, (2016)

Finalmente, se muestra el escenario básico de la generación de la métrica DIT (Ilustración 23), en donde, el actor interno llamado Jenkins ejecuta (por cada componente de un proyecto

Java) el llamado al método *analyse()* de la clase ***CkMoodSensor***; lo primero que hace la clase ***CkMoodSensor*** es crear una instancia de la clase ***DesignDITDecorator***, y posteriormente ejecuta el método *generatesMetrics()*, éste método es el encargado de agregar funcionalidad de forma dinámica para calcular el valor de una métrica en específico, dicho valor es devuelto a la clase ***CkMoodSensor***; finalmente la clase ***CkMoodSensor*** ejecuta el método *saveMeasure()* de la clase ***SensorContext***, el cual es el encargado de almacenar el valor de la métrica calculada en la base de datos.

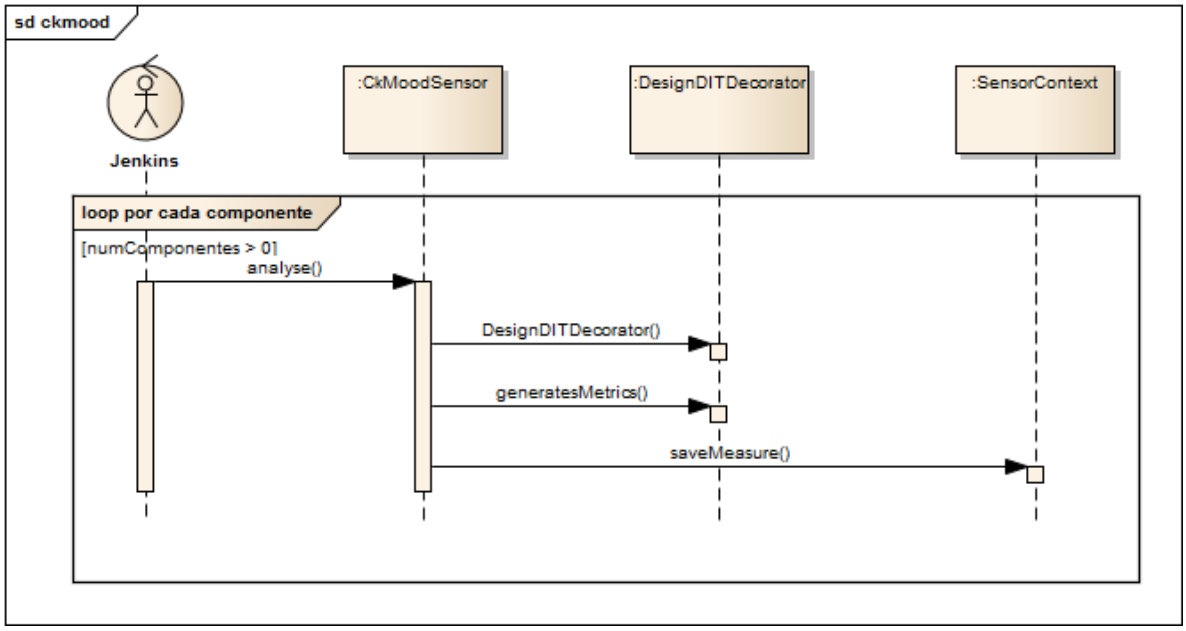


Ilustración 23 Escenario Básico de la generación de la métrica DIT
Elaborada por Romero Ricardo, (2016)

4.3.3 Caso de uso

El tercer artefacto, es el caso de uso. En este artefacto se explica a detalle cada uno de los pasos para lograr su objetivo; se detallan los actores involucrados en su ejecución; se especifican pre y post condiciones que debe de cumplir el caso de uso; se especifican los flujos alternativos, los escenarios clave y en caso de existir los requerimientos especiales.

Para el caso de uso CU1 - Obtener métricas, el actor involucrado es Jenkins <<*internal worker*>>; la pre condición para ejecutar este caso de uso es: el *plugin* CKMOOD se encuentre instalado; el flujo básico de eventos es:

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, ejecuta el análisis de calidad de un proyecto *Java* en específico

Paso 2.- Obtiene las métricas CK por cada componente del proyecto *Java* (ejecuta el CU2)

Paso 3.- Obtiene las métricas MOOD por cada componente del proyecto *Java* (ejecuta el CU3)

Paso 4.- Presenta un resumen de las métricas obtenidas

Paso 5.- Fin del caso de uso

Finalmente, como post condición tenemos que las métricas obtenidas por cada componente son almacenadas en la base de datos.

Para el caso de uso CU2 – Obtener métricas CK, el actor involucrado es Jenkins <<*internal worker*>>; no existen pre condiciones; el flujo básico de eventos es:

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, ejecuta el análisis de calidad de un proyecto *Java* en específico y por cada componente de dicho proyecto realizará los siguientes pasos

Paso 2.- Obtener WMC del componente: La fórmula es

$$WMC = \sum_{i=1}^{n} C_i$$

donde se obtienen el número de métodos de cada componente

Paso 3.- Se almacena la métrica WMC obtenida del componente

Paso 4.- Obtener DIT del componente: Esta métrica es la profundidad en el árbol de herencia de objetos

Paso 5.- Se almacena la métrica DIT obtenida del componente

Paso 6.- Obtener NOC del componente: El NOC de una clase es el número de subclases subordinadas en una jerarquía de objetos

Paso 7.- Se almacena la métrica NOC obtenida del componente

Paso 8.- Obtener CBO del componente: El CBO de una clase, es el número de clases con la que está acoplada, esto es, si usa sus métodos o variables de instancia

Paso 9.- Se almacena la métrica CBO obtenida del componente

Paso 10.- Obtener RFC del componente: El RFC de una clase, es el conjunto de métodos que se pueden ejecutar como respuesta de un mensaje recibido por un objeto. Está dado por la formula

$$RFC = |RS|$$

Donde

$$RS = \{M\} \cup all\ i\ \{R_i\},$$

$\{R_i\}$ es el conjunto de métodos invocados por el método i y $\{M\}$ es el conjunto de todos los métodos de la clase

Paso 11.- Se almacena la métrica RFC obtenida del componente

Paso 12.- Obtener LCOM del componente: LCOM está definido por:

Sea $\{I_j\}$ el conjunto de variables de instancia usados por el método M_i, existen n de esos conjuntos $\{I_1, I_2, \dots, I_n\}$, tal que

$$P = \{(I_i,\ I_j) \vee I_i \cap I_j = 0\} \text{ y } Q = \{(I_i,\ I_j) \vee I_i \cap I_j \neq 0\}$$

entonces:

$$LCOM = |P| - |Q|, si\ |P| > |Q|$$

de otra forma

$$LCOM = 0$$

Paso 13.- Se almacena la métrica LCOM obtenida del componente

Paso 14.- Obtener LCOM1 del componente: Esta métrica es el número de conjuntos disjuntos de métodos locales en una clase

Paso 15.- Se almacena la métrica LCOM1 obtenida del componente

Paso 16.- Obtener MPC del componente: La métrica MPC es el número de métodos invocados en una clase

Paso 17.- Se almacena la métrica MPC obtenida del componente

Paso 18.- Obtener DAC del componente: El DAC de un clase es el número de atributos de una clase que tiene como tipo a otra clase

Paso 19.- Se almacena la métrica DAC obtenida del componente

Paso 20.- Obtener SIZE2 del componente: Esta métrica es el número de atributos más el número de métodos de una clase

Paso 21.- Se almacena la métrica SIZE2 obtenida del componente

Paso 22.- Obtener LCOM2 del componente: Esta variante de la métrica LCOM se obtiene por medio de la siguiente formula

$$LCOM2 = \frac{(a - kl)}{(l - kl)}$$

donde l = Número de atributos; k = Número de métodos y a = Sumatoria de los distintos atributo accedidos por cada método

Paso 23.- Se almacena la métrica LCOM2 obtenida del componente

Paso 24.- Fin del caso de uso

En el caso de uso CU3 – Obtener métricas MOOD, el actor involucrado es Jenkins <<*internal worker*>>; no existen pre condiciones; el flujo básico de eventos es:

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, ejecuta el análisis de calidad de un proyecto *Java* en específico y por cada componente de dicho proyecto realizará los siguientes pasos

Paso 2.- Obtener PF del componente: La fórmula es:

$$PF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) * DC(C_i)]}$$

Donde:

$M_0(C_i) = M_n(C_i) + M_0(C_i)$
$M_n(C_i)$ = Número de métodos definidos en la clase C_i
$M_0(C_i)$ = Número de métodos sobrecargados de la clase C_i
$DC(C_i)$ = Número de descendientes (hijos) de la clase C_i
TC = Total de clases

Paso 3.- Se almacena la métrica PF obtenida del componente

Paso 4.- Obtener CF del componente: Esta métrica se obtiene por

$$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_{client}(C_I, C_J) \right]}{TC}$$

Donde:

$is_{client}(C_i, C_j)$ es la relación entre la clase cliente y la clase objeto
TC = total de clases

Paso 5.- Se almacena la métrica CF obtenida del componente

Paso 6.- Obtener MHF del componente: Se obtiene por medio de la formula

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Donde:

$M_d(C_i) = M_v(C_i) + M_h(C_i)$
$M_d(C_i)$ = Número de métodos definidos en la clase C_i
$M_v(C_i)$ = Número de métodos visibles de la clase C_i
$M_h(C_i)$ = Número de métodos ocultos de la clase C_i
TC = Total de clases

Paso 7.- Se almacena la métrica MHF obtenida del componente

Paso 8.- Obtener AHF del componente: El AHF se obtiene por la formula

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Donde:

$A_d(C_i) = A_v(C_i) + A_h(C_i)$
$A_d(C_i)$ = Número de atributos definidos en la clase C_i
$A_v(C_i)$ = Número de atributos visibles de la clase Ci
$A_h(C_i)$ = Número de atributos ocultos de la clase C_i
TC = Total de clases

Paso 9.- Se almacena la métrica AHF obtenida del componente

Paso 10.- Obtener MIF del componente: Se obtiene por la formula

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Donde:

$M_a(C_i) = M_d(C_i) + M_i(C_i)$
$M_a(C_i)$ = Número de métodos definidos en la clase C_i
$M_d(C_i)$ = Número de métodos declarados de la clase C_i
$M_i(C_i)$ = Número de métodos heredados de la clase C_i
TC = Total de clases

Paso 11.- Se almacena la métrica MIF obtenida del componente

Paso 12.- Obtener AIF del componente: Para ésta métrica se aplica la formula

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Donde:

Aa(Ci) = Ad(Ci) + Ai(Ci)
Aa(Ci) = Número de atributos definidos en la clase Ci
Ad(Ci) = Número de atributos declarados de la clase Ci
Ai (Ci) = Número de atributos heredados de la clase Ci
TC = Total de clases

Paso 13.- Se almacena la métrica AIF obtenida del componente

Paso 14.- Fin del caso de uso

4.3.4 Plan de pruebas

El cuarto artefacto, es el plan de pruebas (PP). En dicho artefacto se describe el alcance, la estrategia general, los recursos y el calendario de las actividades de pruebas que se realizarán a los componentes del proyecto; además se identifican los elementos de prueba, la funcionalidad que será probada, las tareas de pruebas, los responsables de realizarlas y los posibles riesgos que requiera de un plan de contingencia; adicionalmente, éste artefacto sirve de guía a los ingenieros de software, para que realicen la identificación y generación de casos de prueba de forma clara y concreta, con el fin de asegurar que todos los requerimientos del proyecto sean cubiertos y validados.

Las pruebas de software evalúan un producto para determinar que cumple con el objetivo previsto, por lo que es necesario diseñar un plan de pruebas que se adapte y sea coherente con la metodología de desarrollo, que proporcione un enfoque de fácil acceso a la estructura para verificar los requisitos y cuantificar su rendimiento, y que identifique las diferencias entre los resultados previstos y los reales (errores, fallas o desviaciones); es el proceso por medio del cual se evalúa la correcta interpretación y aplicación de los requisitos especificados.

El propósito del artefacto PP es el de proporcionar un plan de pruebas general así como los criterios y casos de pruebas que logren asegurar, en su totalidad, el correcto desempeño y funcionalidad de cada uno de los componentes, verificando que se estén cubriendo los requerimientos para el proyecto, utilizando para ello desde criterios simples y rutinarios, hasta los casos más complejos y de menor incidencia.

Entre los alcances del PP tenemos: proporcionará criterios tanto de operación como técnicos; considerará cada uno de los componentes del proyecto como elementos para las pruebas unitarias, pruebas de integración y pruebas finales de sistema que se realizaran en las instalaciones y con la infraestructura del programador; la metodología de pruebas y este documento permitirá al equipo de profesionales que participan en las pruebas del proyecto, evaluar aspectos como: la lógica estructural, la seguridad, la interconexión, el soporte conceptual, las herramientas de apoyo y sobretodo la independencia de aspectos técnicos del desarrollo de la solución tecnológica contratada, tales como: la plataforma tecnológica o la arquitectura de la solución a probar.

La estrategia general de pruebas indica que en el proyecto se realizarán pruebas incrementales, es decir, en las pruebas unitarias evaluaremos casos propios de la funcionalidad de cada componente, en las pruebas de integración verificaremos únicamente la interacción con otros componentes del sistema y en las pruebas de sistema verificaremos el flujo completo que lleva a cabo el sistema, para llevar a cabo dichas pruebas se hará uso de algunas herramientas, entre las cuales tenemos: *JUnit* es una librería que permite realizar la ejecución de clases *Java* de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera, ésta herramienta se utiliza para la ejecución de pruebas unitarias (PU), pruebas de integración (PI) y pruebas de sistema (PS); la herramienta *Jenkins* es usada como el software de integración continua para el desarrollo de software; *SonarQube* es una herramienta para evaluar el código fuente del componente a realizar.

Finalmente en el PP, también se definen los tipos de pruebas a realizar, entre las cuales tenemos las pruebas unitarias (pruebas que tienen como objetivo verificar la funcionalidad y estructura de cada componente del sistema una vez que ha sido codificado), de integración

(el objetivo de las pruebas de integración es verificar el correcto ensamblaje entre los distintos módulos que componen la solución una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces internas y externas, que cubren la funcionalidad establecida), de sistema (pruebas que buscan diferencias entre la solución desarrollada y los requerimientos, enfocándose en la identificación de los errores que se puedan generar entre la especificación funcional y el diseño del sistema, así como, el negocio objeto de la aplicación) y de regresión (en esta prueba se valida que el sistema mantenga su correcta funcionalidad después de la incorporación de un ajuste, corrección o nuevo requerimiento).

4.3.5 Caso de prueba

El quinto artefacto, es el caso de prueba, en este artefacto se especifica los datos que serán utilizados como entrada para ejecutar el software a probar. Concretamente, el caso de prueba determina un conjunto de entradas, condiciones de ejecución y resultados esperados para un objetivo particular. El resultado de cada caso de prueba determinará si existe una falla en el componente a probar. Para este trabajo de investigación se realizaron 20 pruebas unitarias, 1 prueba de integración y 2 pruebas funcionales de sistema.

4.3.5.1 Pruebas unitarias

Las 20 pruebas unitarias fueron pruebas controladas por datos, se usó la librería *JUnit* y se realizaron para comprobar el correcto cálculo de las métricas CK y MOOD, así como de la funcionalidad importante del módulo; las pruebas unitarias controladas por datos son pruebas que se ejecuta repetidamente para cada origen de datos, en este caso será por cada clase de un proyecto *Java* con el fin de obtener las métricas CK para cada clase del proyecto y para obtener las métricas MOOD de todo el proyecto.

Para llevar a cabo las pruebas unitarias, se creó un archivo llamado jarTest.jar, el cual está compuesto por 11 clases distribuidas en 2 carpetas, en la Ilustración 24, se presenta el diagrama de clases de una de las carpetas y sus relaciones entre las clases que lo componen.

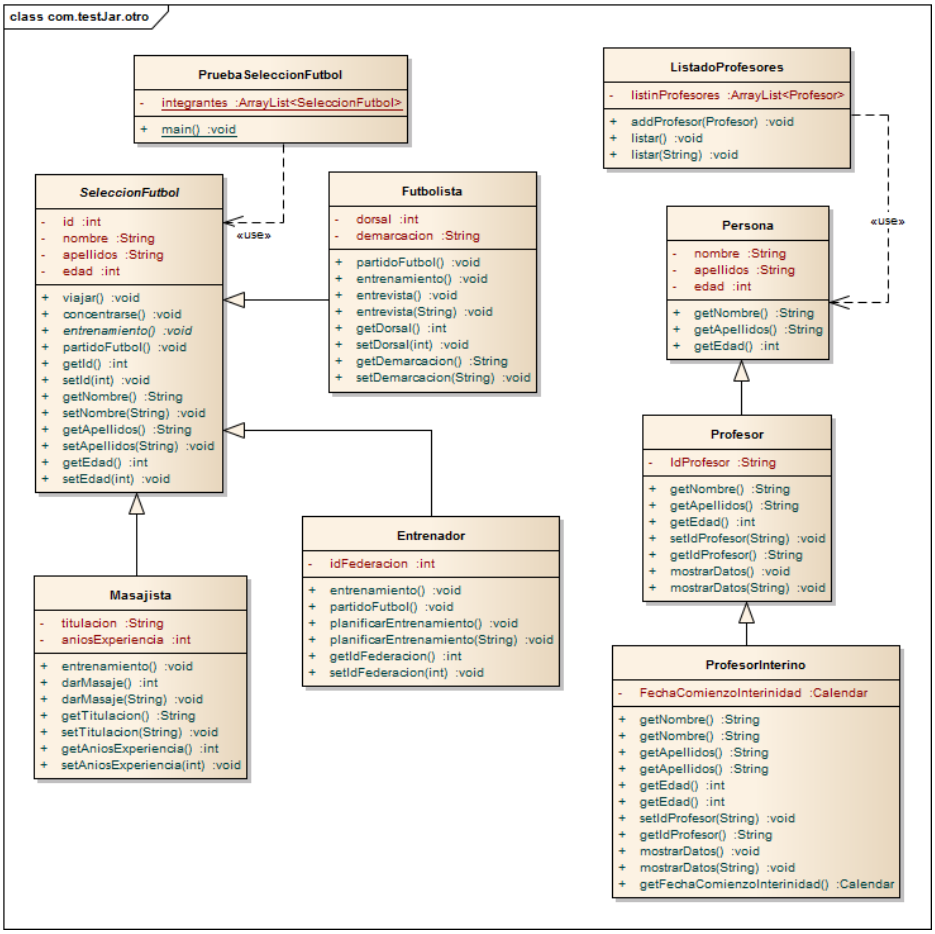


Ilustración 24 Diagrama de clases del archivo jarTest.jar
Elaborada por Romero Ricardo, (2016)

Los valores de las métricas se obtuvieron de forma manual para poder evaluar los posibles valores que se esperan en las pruebas unitarias, y se especifican en casos de prueba específicos, como se puede ver en la Ilustración 25, el caso de prueba para obtener la métrica *WMC*, la cual se aplica a cada clase del proyecto analizado; en la Ilustración 26, se observa el caso de prueba para obtener la métrica PF, que a diferencia de la métrica anterior, ésta métrica se obtiene por todo el proyecto analizado. En ambos casos de prueba se especifican los valores esperados de la prueba, pre y post condiciones para que la prueba se considere exitosa, así como los datos requeridos para cada una de las pruebas.

TestCase01-testGetWmc:

Descripción: Este caso de prueba es para obtener la métrica WMC de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	14
com.testJar.demo.EjemploReflection	1
com.testJar.otro.Entrenador	6
com.testJar.otro.Futbolista	8
com.testJar.otro.ListadoProfesores	3
com.testJar.otro.Masajista	7
com.testJar.otro.Persona	3
com.testJar.otro.Profesor	4
com.testJar.otro.ProfesorInterino	3
com.testJar.otro.PruebaSeleccionFutbol	1
com.testJar.otro.SeleccionFutbol	12

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

Ilustración 25 Caso de Prueba testGetWmc
Elaborada por Romero Ricardo, (2016)

TestCase12-testGetPf:

Descripción: Este caso de prueba es para obtener la métrica PF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0.9354

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

Ilustración 26 Caso de Prueba testGetPf
Elaborada por Romero Ricardo, (2016)

Finalmente, al ejecutar las pruebas controladas (ver Ilustración 27) mediante la librería *JUnit* en el ambiente de desarrollo nos presenta los resultados obtenidos, el número de pruebas unitarias ejecutadas, el tiempo en que cada prueba unitaria consumió al ser ejecutada y el número de pruebas unitarias fallidas.

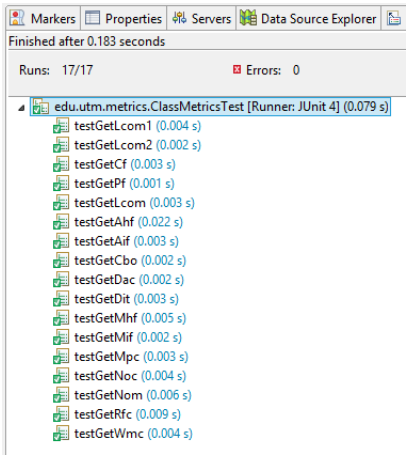
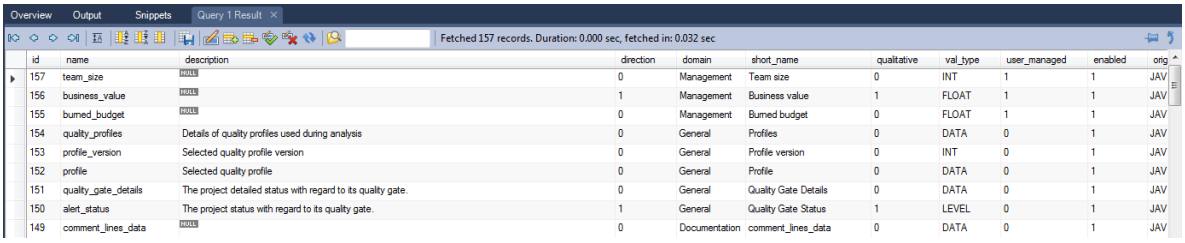


Ilustración 27 Resultado de pruebas unitarias
Elaborada por Romero Ricardo, (2016)

4.3.5.2 Pruebas de integración

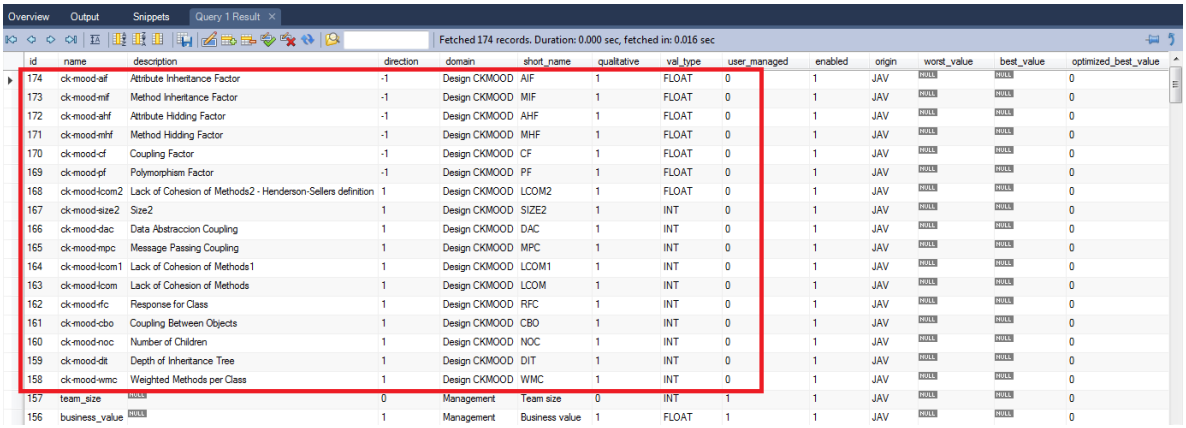
La prueba de integración, se realiza en el ambiente de desarrollo, la cual consiste en la obtención de las métricas CK y MOOD, así como la generación de los 3 reportes que genera el componente construido. Esta prueba se compone de 2 partes: lo referente a la base de datos para realizar el registro de las métricas y el registro de los valores obtenidos de las métricas posterior al análisis realizado, en la Ilustración 28 se presenta la tabla "*sonar.metrics*" que contiene las métricas definidas por la herramienta, como no se ha instalado el componente creado, las métricas de esta investigación no se encuentran registradas.



id	name	description	direction	domain	short_name	qualitative	val_type	user_managed	enabled	orig
157	team_size	NULL	0	Management	Team size	0	INT	1	1	JAV
156	business_value	NULL	1	Management	Business value	1	FLOAT	1	1	JAV
155	burned_budget	NULL	0	Management	Burned budget	0	FLOAT	1	1	JAV
154	quality_profiles	Details of quality profiles used during analysis	0	General	Profiles	0	DATA	0	1	JAV
153	profile_version	Selected quality profile version	0	General	Profile version	0	INT	0	1	JAV
152	profile	Selected quality profile	0	General	Profile	0	DATA	0	1	JAV
151	quality_gate_details	The project detailed status with regard to its quality gate.	0	General	Quality Gate Details	0	DATA	0	1	JAV
150	alert_status	The project status with regard to its quality gate.	1	General	Quality Gate Status	1	LEVEL	0	1	JAV
149	comment_lines_data	NULL	0	Documentation	comment_lines_data	0	DATA	0	1	JAV

Ilustración 28 Catálogo de Métricas antes de instalar el plugin
Elaborada por Romero Ricardo, (2016)

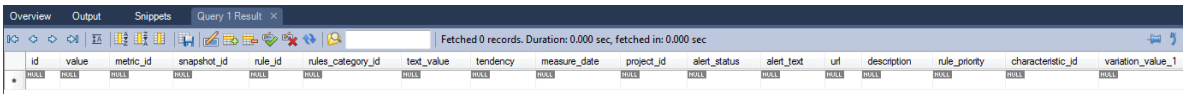
En la Ilustración 29, vemos la misma tabla "*sonar.metrics*", que a diferencia de la ilustración anterior, en dicha ilustración se muestra la información de las métricas, una vez que el componente ya se ha instalado. En dicha ilustración se presentan las métricas CK y MOOD ya registradas en la base de datos.



id	name	description	direction	domain	short_name	qualitative	val_type	user_managed	enabled	origin	worst_value	best_value	optimized_best_value
174	ck-mood-aif	Attribute Inheritance Factor	-1	Design CKMOOD	AIF	1	FLOAT	0	1	JAV	NULL	NULL	0
173	ck-mood-mif	Method Inheritance Factor	-1	Design CKMOOD	MIF	1	FLOAT	0	1	JAV	NULL	NULL	0
172	ck-mood-ahf	Attribute Hidding Factor	-1	Design CKMOOD	AHF	1	FLOAT	0	1	JAV	NULL	NULL	0
171	ck-mood-mhf	Method Hidding Factor	-1	Design CKMOOD	MHF	1	FLOAT	0	1	JAV	NULL	NULL	0
170	ck-mood-cf	Coupling Factor	-1	Design CKMOOD	CF	1	FLOAT	0	1	JAV	NULL	NULL	0
169	ck-mood-pf	Polymorphism Factor	-1	Design CKMOOD	PF	1	FLOAT	0	1	JAV	NULL	NULL	0
168	ck-mood-lcom2	Lack of Cohesion of Methods2 - Henderson-Sellers definition	1	Design CKMOOD	LCOM2	1	FLOAT	0	1	JAV	NULL	NULL	0
167	ck-mood-size2	Size2	1	Design CKMOOD	SIZE2	1	INT	0	1	JAV	NULL	NULL	0
166	ck-mood-dac	Data Abstraccion Coupling	1	Design CKMOOD	DAC	1	INT	0	1	JAV	NULL	NULL	0
165	ck-mood-mpc	Message Passing Coupling	1	Design CKMOOD	MPC	1	INT	0	1	JAV	NULL	NULL	0
164	ck-mood-lcom1	Lack of Cohesion of Methods1	1	Design CKMOOD	LCOM1	1	INT	0	1	JAV	NULL	NULL	0
163	ck-mood-lcom	Lack of Cohesion of Methods	1	Design CKMOOD	LCOM	1	INT	0	1	JAV	NULL	NULL	0
162	ck-mood-rfc	Response for Class	1	Design CKMOOD	RFC	1	INT	0	1	JAV	NULL	NULL	0
161	ck-mood-cbo	Coupling Between Objects	1	Design CKMOOD	CBO	1	INT	0	1	JAV	NULL	NULL	0
160	ck-mood-noc	Number of Children	1	Design CKMOOD	NOC	1	INT	0	1	JAV	NULL	NULL	0
159	ck-mood-dit	Depth of Inheritance Tree	1	Design CKMOOD	DIT	1	INT	0	1	JAV	NULL	NULL	0
158	ck-mood-wmc	Weighted Methods per Class	1	Design CKMOOD	WMC	1	INT	0	1	JAV	NULL	NULL	0
157	team_size	NULL	0	Management	Team size	0	INT	1	1	JAV	NULL	NULL	0
156	business_value	NULL	1	Management	Business value	1	FLOAT	1	1	JAV	NULL	NULL	0

Ilustración 29 Catálogo de Métricas después de instalar el plugin
Elaborada por Romero Ricardo, (2016)

En la tabla "*sonar.project_measures*" se almacena la información obtenida de las métricas por proyecto una vez que se haya realizado el análisis, en la Ilustración 30 se muestra que dicha tabla se encuentra vacía debido a que no se ha realizado ningún análisis.



id	value	metric_id	snapshot_id	rule_id	rules_category_id	text_value	tendency	measure_date	project_id	alert_status	alert_text	url	description	rule_priority	characteristic_id	variation_value_1

Ilustración 30 Métricas obtenidas antes de realizar el análisis
Elaborada por Romero Ricardo, (2016)

En la Ilustración 31, se presenta la tabla "*sonar.project_measures*", después de haber realizado el análisis de un proyecto *Java* por la herramienta. La información importante que se almacena en la tabla es: la referencia de la métrica (columna "*metric_id*"), el valor de la métrica obtenida por el análisis (columna "*value*") y el valor del archivo o proyecto analizado (columna "*url*").

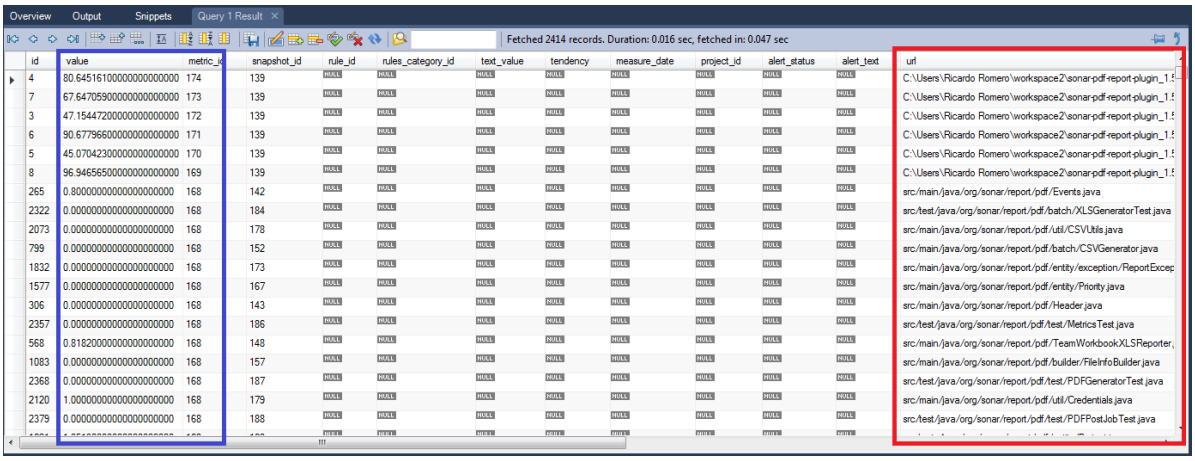


Ilustración 31 Métricas obtenidas después de realizar el análisis
Elaborada por Romero Ricardo, (2016)

En relación a la interfaz gráfica, En la Ilustración 32, vemos la interfaz gráfica de la herramienta *SonarQube* antes de realizar prueba alguna, dicha interfaz es la página principal que consta de 2 secciones, en la primera se muestra el nombre del proyecto, la versión del proyecto, el número de líneas de código, la deuda técnica y la fecha y hora del último análisis realizado al proyecto. La segunda sección presenta los proyectos en forma gráfica tomando en cuenta el número de líneas de código de cada proyecto.

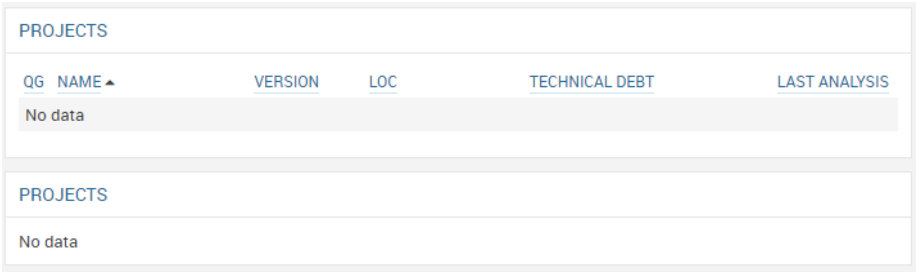


Ilustración 32 Interfaz gráfica antes de realizar el análisis
Elaborada por Romero Ricardo, (2016)

En la Ilustración 33 vemos la misma interfaz gráfica de la ilustración anterior, pero con la única diferencia que el análisis ya fue realizado, donde el proyecto analizado se llama SonarQube PDF Report, el cual tiene 3,805 líneas de código y es el único proyecto en la herramienta.

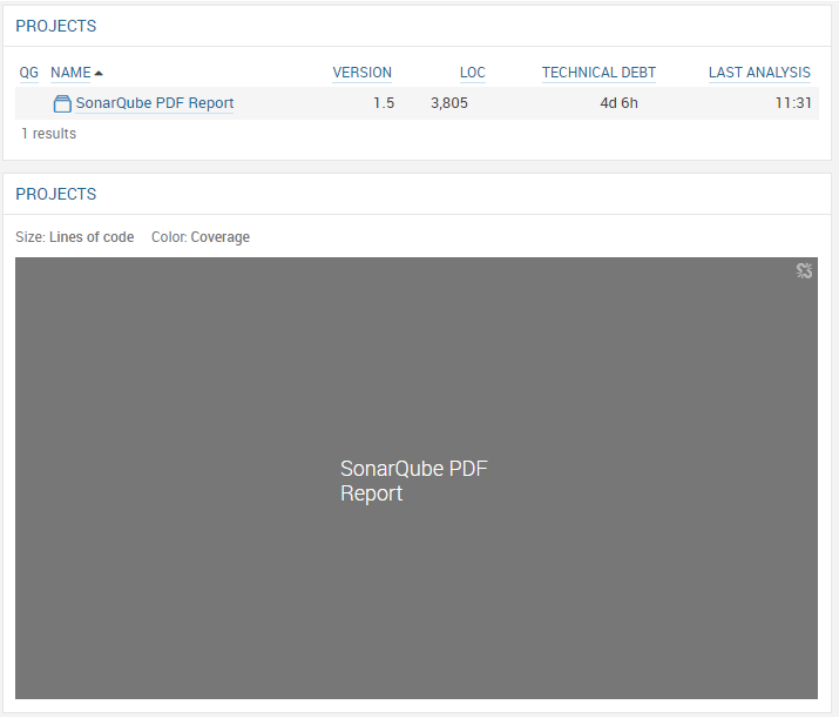


Ilustración 33 Interfaz gráfica general después de realizar el análisis
Elaborada por Romero Ricardo, (2016)

En la Ilustración 34 vemos la información del análisis realizado a un proyecto en específico, la sección que se encuentra enmarcada en rojo, muestra la interfaz gráfica del componente desarrollado, como se puede ver en la ilustración, se presentan las opciones para descargar el resultado del análisis en tres formatos diferentes (*PDF*, *XLS* y *CSV*).

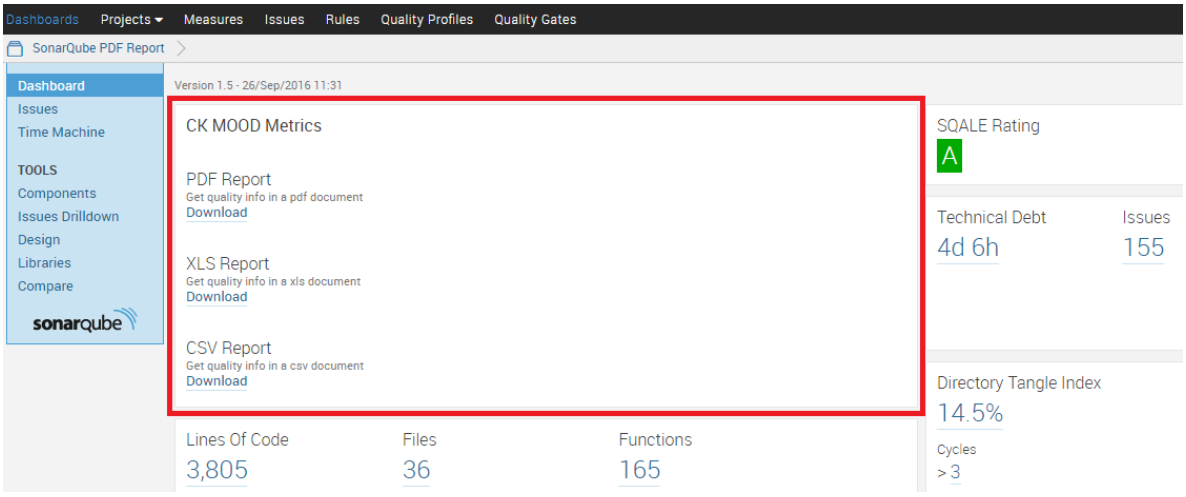


Ilustración 34 Interfaz gráfica detallada después de realizar el análisis
Elaborada por Romero Ricardo, (2016)

4.3.5.1 Pruebas Funcionales

La primera prueba funcional de sistema, se aplica al producto final en un ambiente pre productivo, debido a que la prueba se compone de diversas herramientas, se irán presentando imágenes para demostrar el o los estados de las diversas interfaces gráficas. En la Ilustración 35, vemos la interfaz gráfica de la herramienta *Jenkins*, en donde vemos el nombre del proyecto, la fecha del último análisis exitoso, la fecha del último análisis fallido y la duración del análisis realizado.

Ilustración 35 Servidor de IC antes del análisis
Elaborada por Romero Ricardo, (2016)

Al momento de ejecutar el análisis de un proyecto en específico, se puede tener acceso a la consola de la herramienta, para verificar la información de la ejecución. En la Ilustración 36 se muestran secciones de la bitácora al momento de generar los reportes en sus diferentes formatos, cuando son enviados al servidor, cuando se consultas las métricas. Al final del análisis, se presenta el estado, el tiempo total, la fecha y hora de terminación, así como el tamaño de la memoria usada.

```
[INFO] [12:39:30.477] -> Clean sigafd-cyc-inversion-core [id=290]
[INFO] [12:39:30.487] Executing post-job class edu.utm.sonar.ckmood.report.pdf.batch.PDFPostJob
[INFO] [12:39:30.487] Executing decorator: PDF Report
[INFO] [12:39:30.497] Team workbook report type selected
[INFO] [12:39:30.660] Retrieving project info for com.sigafd:sigafd-cyc-inversion-core
[INFO] [12:39:31.094]     Retrieving measures
[INFO] [12:39:32.380]     Retrieving most violated rules
[INFO] [12:39:40.429]     Retrieving most violated files
[INFO] [12:39:40.638]     Retrieving most complex elements
[INFO] [12:39:40.699]     Retrieving most duplicated files
[INFO] [12:39:40.765]     Retrieving ck Metrics
[INFO] [12:39:42.735] Generating PDF report...
[INFO] [12:39:43.501] PDF report generated (see com.sigafd-sigafd-cyc-inversion-core.pdf on build output directory)
[INFO] [12:39:43.504] Uploading com.sigafd-sigafd-cyc-inversion-core.pdf to server...
[INFO] [12:39:43.773] File uploaded.
[INFO] [12:39:43.892] Generating XLS report...
[INFO] [12:39:43.950] Retrieving project info for com.sigafd:sigafd-cyc-inversion-core
[INFO] [12:39:43.980]     Retrieving measures
[INFO] [12:39:44.635]     Retrieving most violated rules
[INFO] [12:39:45.887]     Retrieving most violated files
[INFO] [12:39:45.957]     Retrieving most complex elements
[INFO] [12:39:46.017]     Retrieving most duplicated files
[INFO] [12:39:46.076]     Retrieving ck Metrics
[INFO] [12:39:47.971] Excel written successfully...........
[INFO] [12:39:47.972] XLS report generated (see com.sigafd-sigafd-cyc-inversion-core.xls on build output directory)
[INFO] [12:39:47.972] Uploading com.sigafd-sigafd-cyc-inversion-core.xls to server...
[INFO] [12:39:48.017] File uploaded.
[INFO] [12:39:48.025] Generating CSV report...
[INFO] [12:39:48.025] Retrieving project info for com.sigafd:sigafd-cyc-inversion-core
[INFO] [12:39:48.089]     Retrieving measures
[INFO] [12:39:48.721]     Retrieving most violated rules
[INFO] [12:39:49.602]     Retrieving most violated files
[INFO] [12:39:49.656]     Retrieving most complex elements
[INFO] [12:39:49.714]     Retrieving most duplicated files
[INFO] [12:39:49.769]     Retrieving ck Metrics
[INFO] [12:39:51.082] CSV report generated (see com.sigafd-sigafd-cyc-inversion-core.csv on build output directory)
[INFO] [12:39:51.082] Uploading com.sigafd-sigafd-cyc-inversion-core.csv to server...
[INFO] [12:39:51.112] File uploaded.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 59.176 s
[INFO] Finished at: 2016-09-26T12:39:51-05:00
[INFO] Final Memory: 44M/815M
```

Ilustración 36 Consola del servidor de IC con el resultado del análisis
Elaborada por Romero Ricardo, (2016)

Finalmente en la Ilustración 37 se presenta la interfaz gráfica de la herramienta *Jenkins* posterior al análisis, en donde nos muestra que el proyecto se analizó en 1 minuto con 51 segundos, seguido de "#1", éste valor nos indica el número de ejecución del análisis al proyecto llamado "sigafd-cyc-inversion-core".

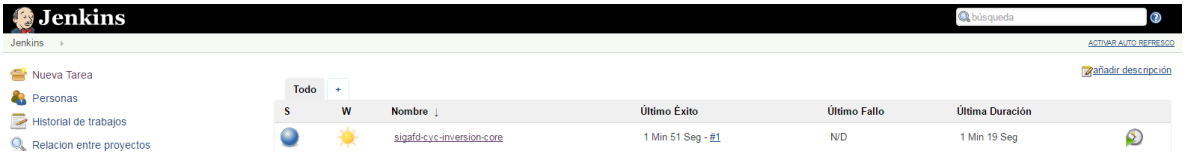


Ilustración 37 Servidor de IC después del análisis
Elaborada por Romero Ricardo, (2016)

Por otro lado, tenemos la herramienta de análisis de código llamada *SonarQube*, en donde se muestra la interfaz gráfica general de la herramienta (ver Ilustración 38), nos presenta los proyectos analizados en forma de lista, así como nos presenta de forma gráfica el tamaño en líneas de código de los proyectos analizados.

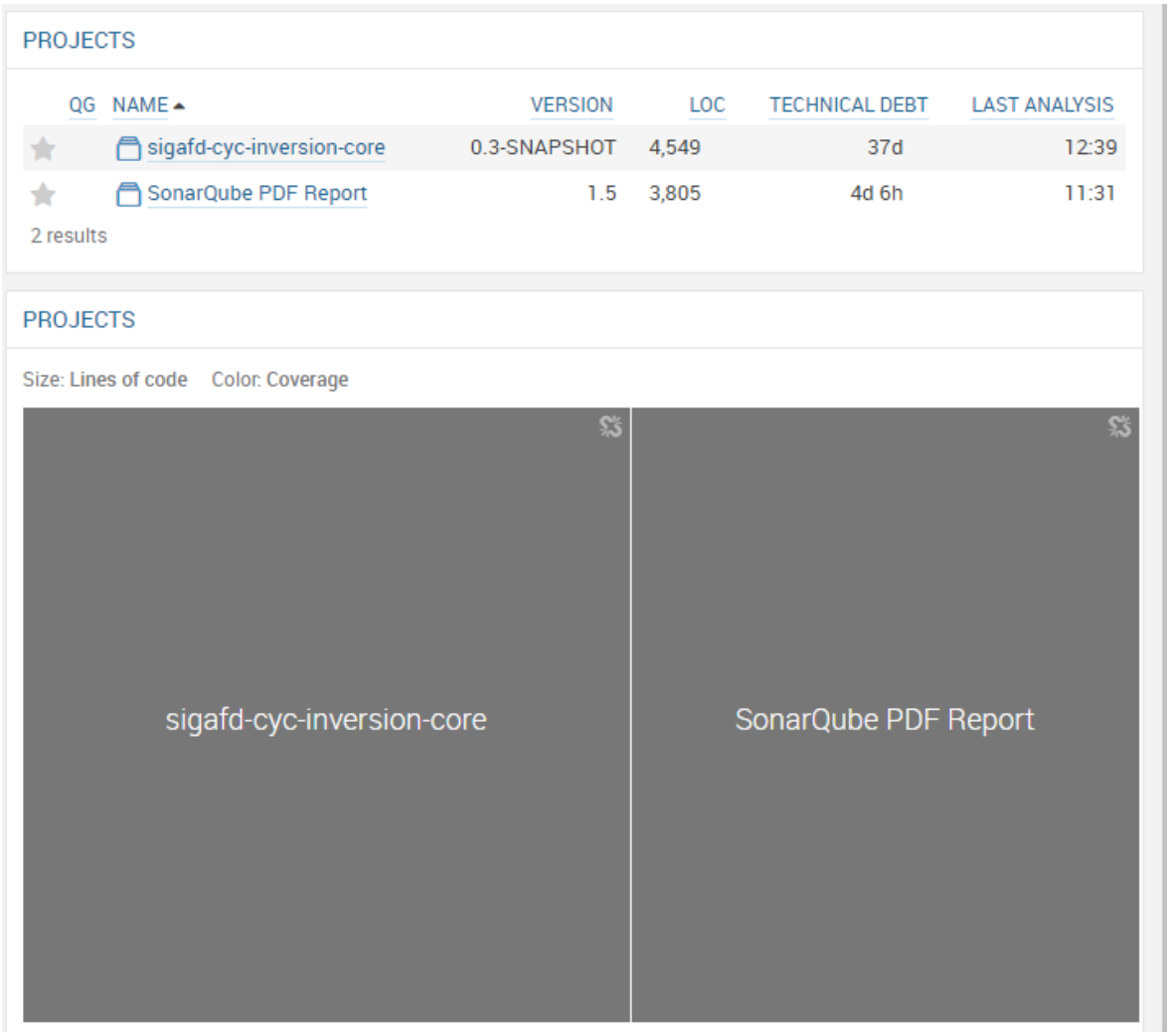


Ilustración 38 Interfaz gráfica general de la herramienta SonarQube
Elaborada por Romero Ricardo, (2016)

Una vez que tenemos la interfaz general, podemos seleccionar un proyecto en específico para ver las métricas obtenidas en el proyecto, de forma más detallada (Ilustración 39), es importante recalcar, que la interfaz detallada presenta todas las métricas k fueron analizadas, incluidas las métricas CK y MOOD. Nos muestra las diversas opciones para realizar la descarga de los 3 reportes generados por el componente construido.

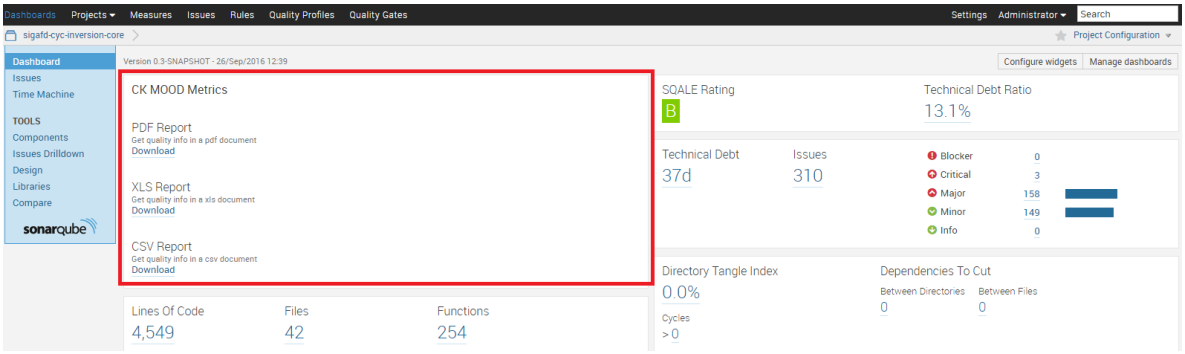
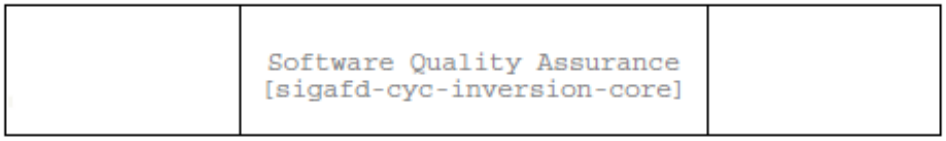


Ilustración 39 Interfaz gráfica detallada del proyecto analizado
Elaborada por Romero Ricardo, (2016)

En la siguiente ilustración, se presenta una muestra del reporte generado en formato PDF (ver Ilustración 40), en donde podemos apreciar entre otras métricas las líneas de código, el porcentaje de comentarios, la complejidad y el número de puntos de decisión del código analizado, así como 5 de las métricas MOOD. Este tipo de reporte es más ejecutivo en relación a los otros formatos, debido a que la información contenida en dicho reporte no puede ser modificada fácilmente.



1. sigafd-cyc-inversion-core

This chapter presents an overview of the project measures. This dashboard shows the most important measures related to project quality, and it provides a good starting point for identifying problems in source code.

1.1. Report Overview

Static Analysis

Lines of code	Comments	Complexity
4,549	16.3%	2.4
N/A Packages	883 Comment lines	14.3 /Class
42 Classes		599 Decision points
254 Methods		
28.4% Duplicated lines		

Dynamic Analysis

Code Coverage	Test Success
N/A	N/A
N/A Tests	N/A Failures
	N/A Errors

Coding Rules Issues

Technical Debt	Issues
37d	310

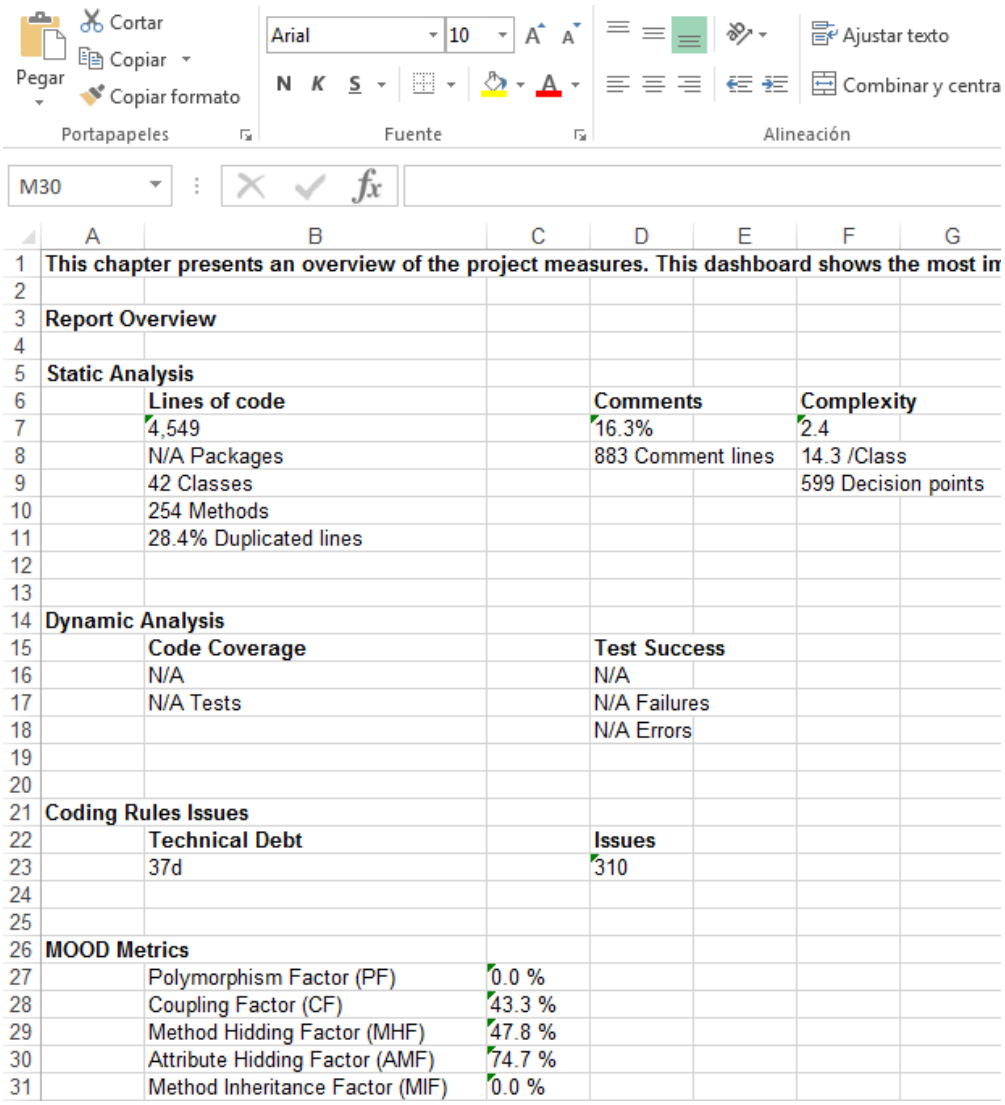
MOOD Metrics

Polymorphism Factor (PF)	0.0 %	*(Optimal Value 0 - 10%)*
Coupling Factor (CF)	43.3 %	*(Optimal Value < 10%)*
Method Hidding Factor (MHF)	47.8 %	*(Optimal Value 12 - 22%)*
Attribute Hidding Factor (AMF)	74.7 %	*(Optimal Value > 75%)*
Method Inheritance Factor (MIF)	0.0 %	*(Optimal Value 60 - 80%)*

Ilustración 40 Reporte del análisis realizado en formato PDF
Elaborada por Romero Ricardo, (2016)

La Ilustración 41, nos presenta el mismo reporte que el anterior, pero en formato XLS. En este tipo de formato, la información contenida en el reporte si puede ser modificada por

alguna herramienta externa que permita manipular archivos. Este tipo de formato es muy usado gracias a la popularidad de Microsoft Excel.

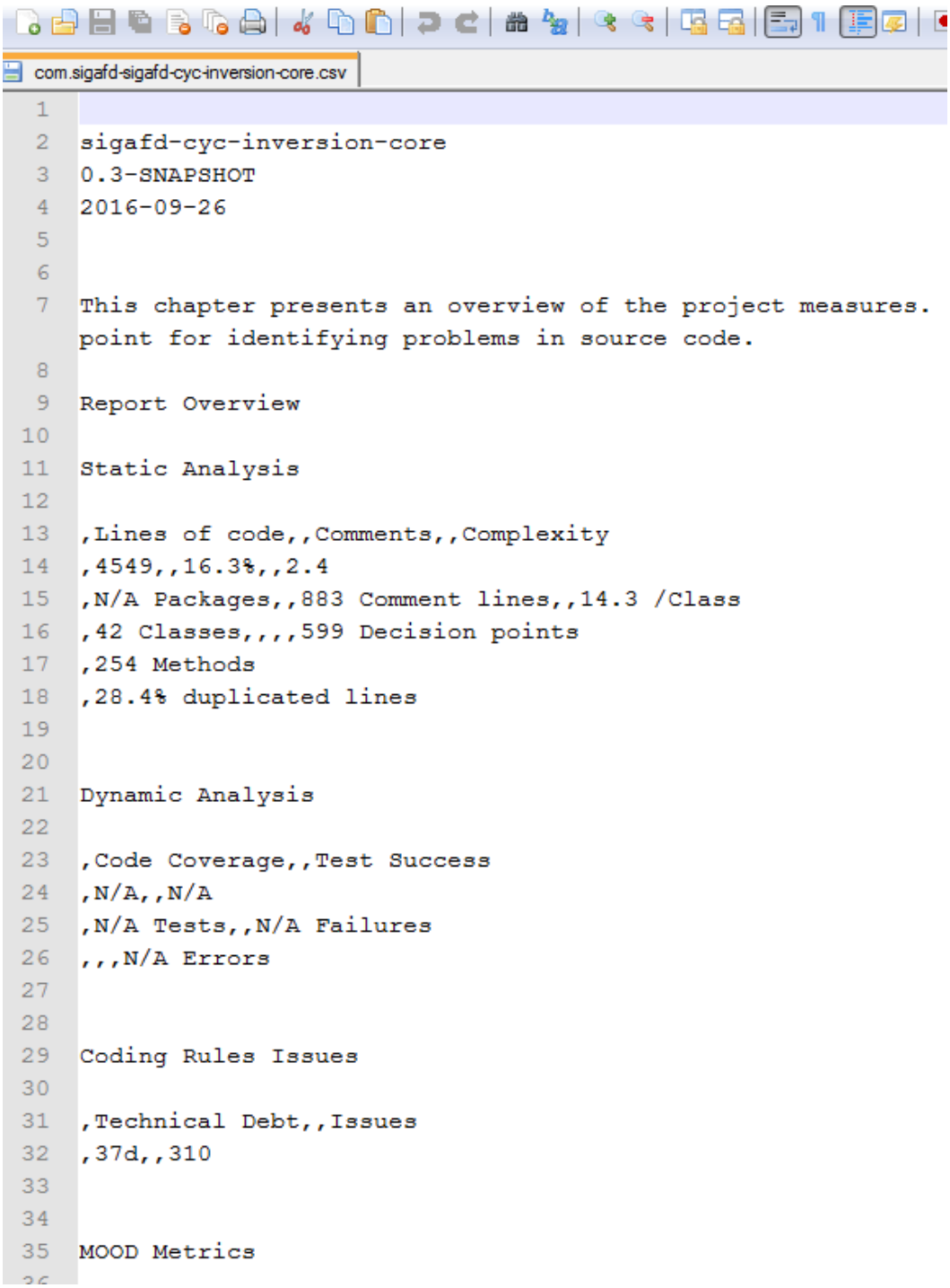


	A	B	C	D	E	F	G
1	**This chapter presents an overview of the project measures. This dashboard shows the most im**						
2							
3	**Report Overview**						
4							
5	**Static Analysis**						
6		**Lines of code**		**Comments**		**Complexity**	
7		4,549		16.3%		2.4	
8		N/A Packages		883 Comment lines		14.3 /Class	
9		42 Classes				599 Decision points	
10		254 Methods					
11		28.4% Duplicated lines					
12							
13							
14	**Dynamic Analysis**						
15		**Code Coverage**		**Test Success**			
16		N/A		N/A			
17		N/A Tests		N/A Failures			
18				N/A Errors			
19							
20							
21	**Coding Rules Issues**						
22		**Technical Debt**		**Issues**			
23		37d		310			
24							
25							
26	**MOOD Metrics**						
27		Polymorphism Factor (PF)	0.0 %				
28		Coupling Factor (CF)	43.3 %				
29		Method Hidding Factor (MHF)	47.8 %				
30		Attribute Hidding Factor (AMF)	74.7 %				
31		Method Inheritance Factor (MIF)	0.0 %				

Ilustración 41 Reporte del análisis realizado en formato XLS
Elaborada por Romero Ricardo, (2016)

Por último, el reporte de las métricas obtenidas por el análisis en formato CSV es mostrado en la Ilustración 42. Los archivos CSV (del inglés *comma-separated values*) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que

las columnas se separan por comas (o punto y coma, entre otros separadores) y las filas por saltos de línea. Este formato de archivos no necesita de herramientas especiales, haciendo que el archivo sea manipulado fácilmente.


com.sigafd-sigafd-cyc-inversion-core.csv

```
1
2  sigafd-cyc-inversion-core
3  0.3-SNAPSHOT
4  2016-09-26
5
6
7  This chapter presents an overview of the project measures.
   point for identifying problems in source code.
8
9  Report Overview
10
11 Static Analysis
12
13 ,Lines of code,,Comments,,Complexity
14 ,4549,,16.3%,,2.4
15 ,N/A Packages,,883 Comment lines,,14.3 /Class
16 ,42 Classes,,,,599 Decision points
17 ,254 Methods
18 ,28.4% duplicated lines
19
20
21 Dynamic Analysis
22
23 ,Code Coverage,,Test Success
24 ,N/A,,N/A
25 ,N/A Tests,,N/A Failures
26 ,,,N/A Errors
27
28
29 Coding Rules Issues
30
31 ,Technical Debt,,Issues
32 ,37d,,310
33
34
35 MOOD Metrics
```

Ilustración 42 Reporte del análisis realizado en formato CSV
Elaborada por Romero Ricardo, (2016)

La segunda prueba funcional de sistema se realiza cuando no se encuentra activado el *plugin* que obtiene las métricas CK – MOOD, en la Ilustración 43, se presenta la interfaz gráfica para activar o desactivar la generación de las métricas CK – MOOD.

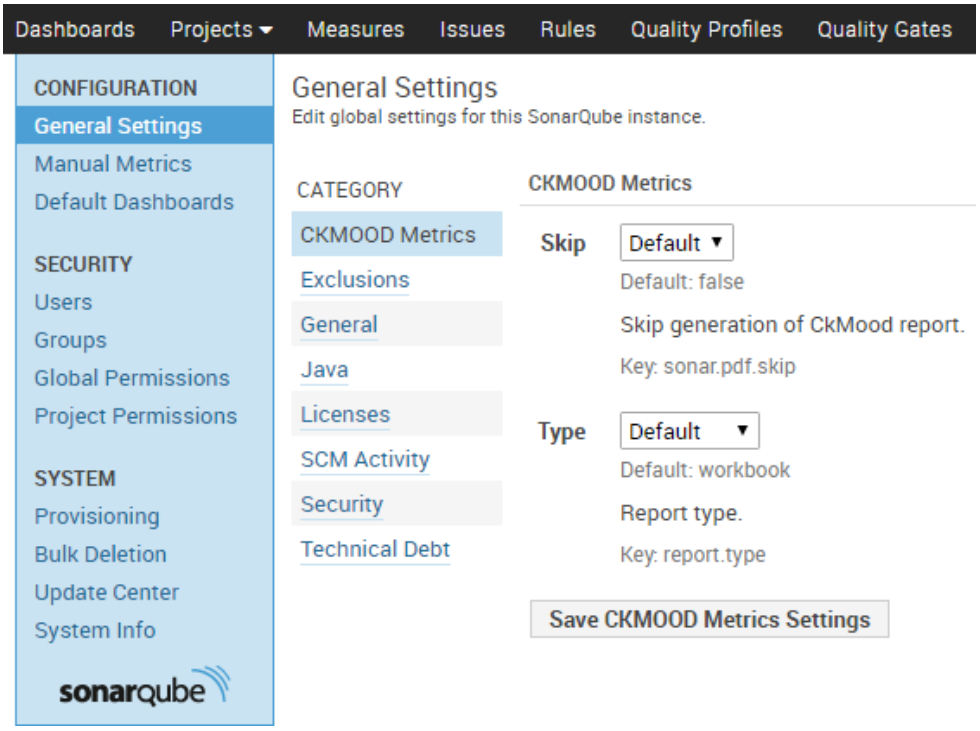


Ilustración 43 Pantalla inicial de la configuración del componente
Elaborada por Romero Ricardo, (2016)

Después de desactivar el análisis de las métricas CK – MOOD, si nuevamente se genera el análisis desde la herramienta *Jenkins*, en la herramienta *SonarQube* al mostrar la información detallada del análisis a un proyecto en específico, no se tiene disponible la descarga del reporte de las métricas CK – MOOD, en su lugar se muestran mensajes en donde especifica que los reportes no fueron generados, posterior al análisis de un proyecto en específico (ver Ilustración 44). La activación y/o desactivación del análisis de las métricas CK – MOOD se realiza a nivel general, esto quiere decir, que no se puede seleccionar el proyecto a analizar, o analiza todos o no analiza nada.

Ilustración 44 Pantalla de resultado con el componente desactivado
Elaborada por Romero Ricardo, (2016)

4.3.6 Código fuente

Finalmente, el último artefacto de ésta fase, es el código fuente. Estos artefactos contienen la lógica necesaria en el lenguaje de programación *Java* para cumplir con la funcionalidad requerida necesaria para satisfacer los requerimientos del proyecto. Por lo general, se pretende que, el código fuente cumpla con las siguientes características: correctitud (un programa es correcto si hace lo que debe hacer tal y como se especificó en su análisis y diseño), claridad (es importante que el código fuente sea lo más claro y legible posible, para facilitar tanto su desarrollo como su posterior mantenimiento) y eficiencia (además de que el código fuente sea correcto, lo debe de hacer gestionando de la mejor forma posible los recursos que utiliza).

La construcción se realizó usando la biblioteca *Apache Commons BCEL*, la cual nos facilita el analizar y manipular los archivos binarios *Java (archivos .class)*, la biblioteca utiliza clases que están representadas por objetos que contienen toda la información de una clase como métodos, campos y las instrucciones. En la Ilustración 45, se presenta el diagrama de secuencia para obtener la métrica DIT.

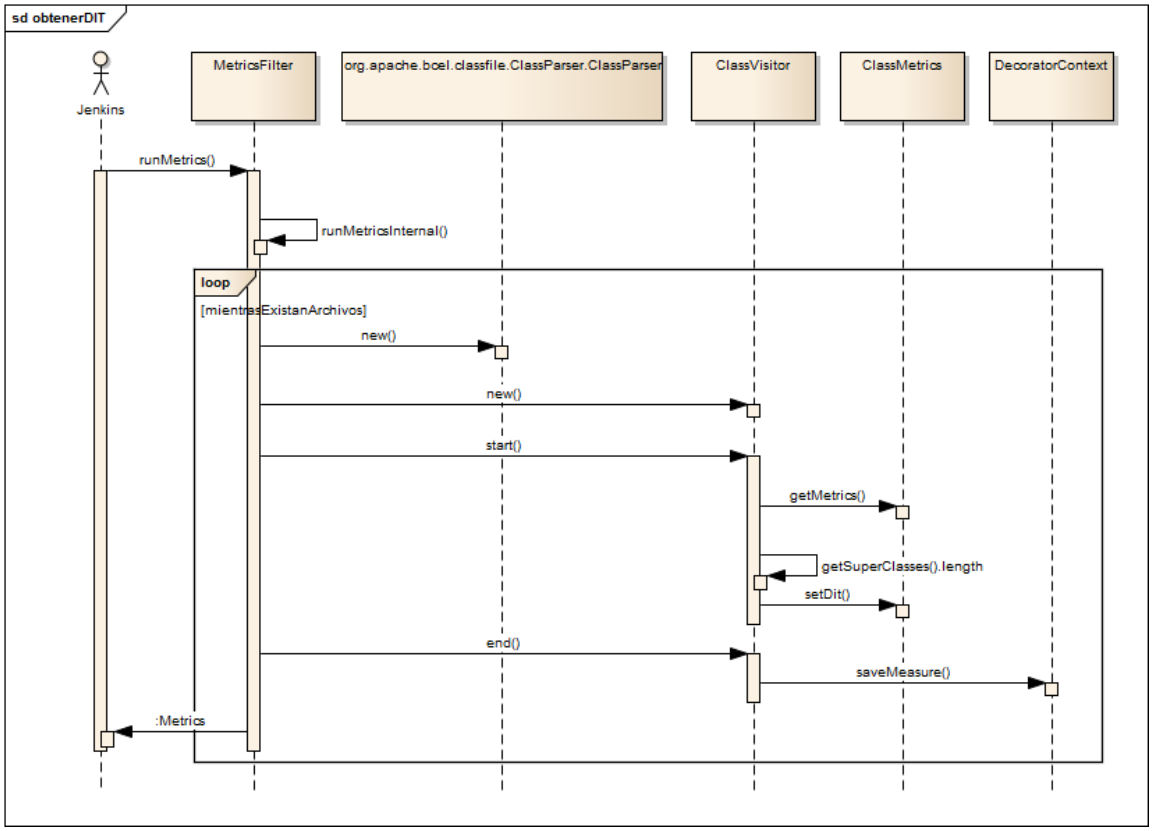


Ilustración 45 Diagrama de Secuencia para obtener la métrica DIT
Elaborada por Romero Ricardo, (2016)

El método *runMetrics()*, de clase *MetricsFilter* recibe como parámetro el archivo *jar* a analizar; el método *runMetricsInternal()* extrae cada uno de los archivos binarios y por cada archivo binario se crea una instancia de la clase *ClassVisitor*; ejecuta el método *start()* de la clase *ClassVisitor* para obtener la definición de las clases al llamar al método *getMetrics()* de la clase *ClassMetrics;* posteriormente ejecuta el método *getSuperClasses().length*, el cual obtiene el número de clases padre del archivo binario analizado; con el valor obtenido, se actualiza la variable DIT; en el método *end()* de la clase *ClassVisitor* se almacena el valor de la métrica en la base de datos, por medio del método *saveMeasure* de la clase *DecoratorContext*; una vez que analiza todos los archivos binarios del *jar* analizado, la clase *MetricsFilter* devuelve el control de la ejecución a la herramienta *Jenkins*.

4.4.- Fase Transición

En la fase de transición se realiza la creación del manual de instalación y la documentación técnica del producto desarrollado. En el APÉNDICE 4, se presentan los artefactos que forman parte de ésta fase, mismos que a continuación se detallarán.

4.4.1 Manual de instalación

El primer artefacto de ésta fase es el manual de instalación del componente desarrollado, en dicho documento se especifica la forma en que se debe de instalar el componente y como se debe de activar.

La instalación del componente es muy simple, sólo basta copiar el archivo sonar-ckmood-plugin-0.1-SNAPSHOT.jar en la carpeta "\extensions\plugins" que se encuentra en el directorio de instalación de la herramienta *SonarQube*.

Una vez instalado, hay que hacer visible la interfaz gráfica del componente, y para llevarlo a cabo, debemos de seguir los siguientes pasos:

1. Iniciamos sesión en la herramienta *SonarQube*, seleccionando la opción "*Log in*"
2. Seleccionamos un proyecto analizado
3. Seleccionamos la opción "*Configure Widgets*"
4. Buscamos el "*CKMOOD report widget*"
5. Seleccionamos la opción "*Add widget*"
6. Finalmente seleccionamos la opción "*Back to dashboard*" para salir de la configuración de widgets

Para realizar la activación o desactivación del componente, debemos de seguir los pasos siguientes:

1. Iniciamos sesión en la herramienta *SonarQube*, seleccionando la opción "*Log in*"
2. Seleccionamos la opción "*Settings*"
3. Del menú "*CONFIGURATION*" seleccionamos la opción "*General Metrics*"
4. Del menú "*CATEGORY*", seleccionamos la opción "*CKMOOD Metrics*"
5. En la opción "*Skip*", es donde se activa o desactiva el análisis de las métricas
6. Después se almacenan los cambios al seleccionar la opción "*Save CKMOOD Metrics Settings*"
7. Finalmente, seleccionamos la opción "*Dashboards*" para regresar a la pantalla principal

4.4.2 Documentación técnica

El segundo artefacto es la documentación técnica del componente, ésta documentación fue creada de forma automática por medio de la herramienta llamada *Javadoc*, la cual genera la documentación de aplicaciones en formato *HTML* a partir del código fuente; es el estándar de la industria para documentar clases de *Java*.

En la Ilustración 46, se muestra la información general del proyecto y entre la información más importante que es presentada, puedo mencionar que nos presenta un resumen del proyecto, las dependencias que tiene el proyecto e información del equipo de trabajo.

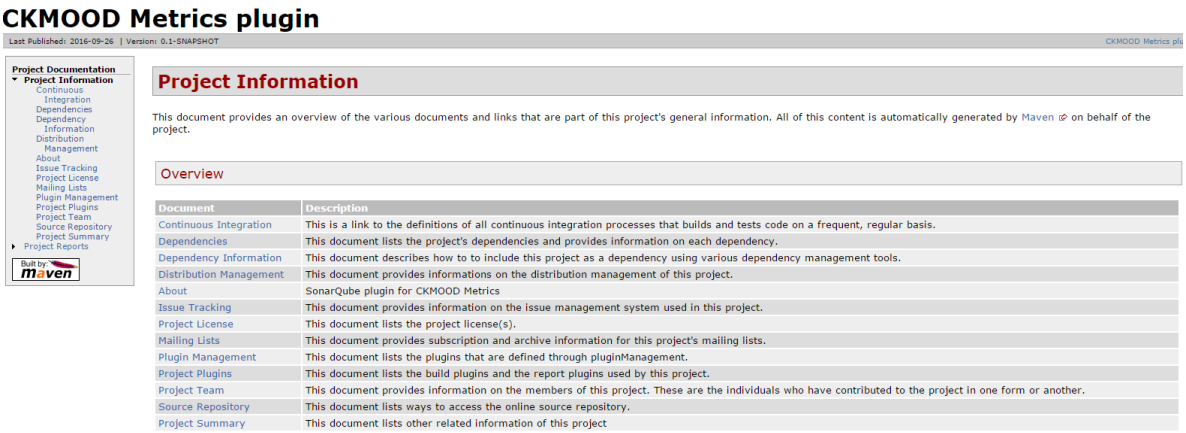


Ilustración 46 Documentación del proyecto
Elaborada por Romero Ricardo, (2016)

En la Ilustración 47, se observa la documentación técnica a nivel de paquete, podemos ver una sección donde nos muestra todos los paquetes, otra sección donde presenta todas las clases y en la parte central vemos la información de un paquete del proyecto en específico.

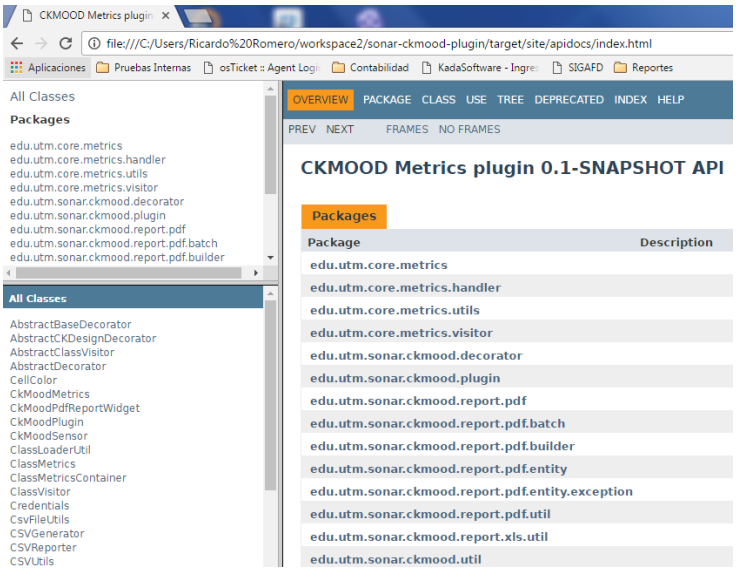


Ilustración 47 Documentación técnica de un paquete
Elaborada por Romero Ricardo, (2016)

Por último, en la Ilustración 48 vemos la documentación técnica a nivel de clase. Toda la información presentada se basa en la documentación creada durante el diseño, tanto en las clases, en los atributos, como en los métodos.

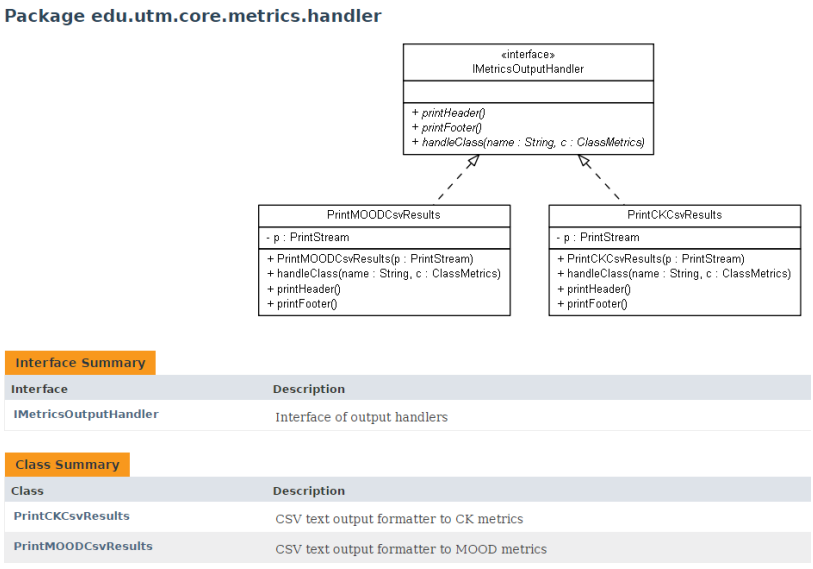


Ilustración 48 Documentación técnica de una clase
Elaborada por Romero Ricardo, (2016)

5.- EVALUACION

Para llevar a cabo la evaluación del componente se utilizaron en total 8 computadoras de escritorio, distribuidas de la siguiente manera: 5 computadoras con *Windows* 7, con la herramienta de control de calidad *SonarQube* y el componente creado *sonar-ckmood-plugin*, un servidor de IC con *Windows* 7 y la herramienta *Jenkins*, un servidor de base de datos *MySQL* instalado sobre *Ubuntu* y finalmente un servidor de código fuente *Subversion* instalado sobre *CentOs*, tal como se muestra en la Ilustración 49. La intención de seleccionar 5 computadoras con *Windows* 7 y *SonarQube* fue la de ejecutar en paralelo el análisis de las herramienta seleccionadas para ser evaluadas.

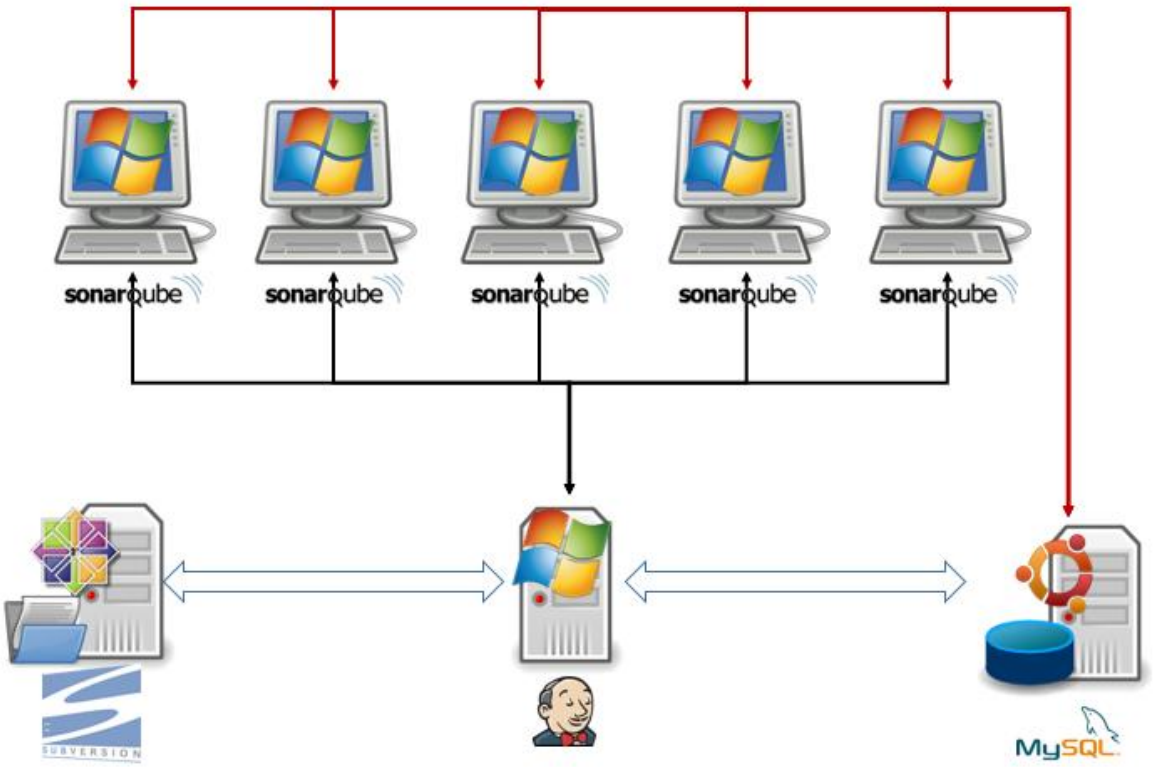


Ilustración 49 Esquema de Ejecución de la Evaluación del Componente
Elaborada por Romero Ricardo, (2016)

La evaluación del componente se realizó a 5 herramientas de la categoría de software libre, creadas en el lenguaje *Java* y con la herramienta *Maven*. Se seleccionaron las cinco herramientas formadas por los proyectos *ActiveMQ*, *Apache Archiva*, *Apache Maven*, *iText Core* y *JFreeChart*, por tener más de 50,000 líneas de código y en la Tabla 6 se muestran sus características.

Herramienta	Versión	Líneas de código	Fecha de descarga
ActiveMQ	5.15.0-SNAPSHOT	203,668	15/10/2016
Apache Archiva	2.2.2-SNAPSHOT	51,227	15/10/2016
Apache Maven	3.4.0-SNAPSHOT	61,350	15/10/2016
iText Core	5.5.11-SNAPSHOT	110,814	15/10/2016
JFreeChart	1.5.0-SNAPSHOT	98,525	15/10/2016

Tabla 6 Características de las herramientas evaluadas
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

Apache ActiveMQ es un agente de mensajería desarrollado por la *Apache Software Foundation* (bajo licencia *Apache* 2.0) que implementa plenamente la especificación de *Java Message Service* 1.1 (*JMS*). Ofrece características empresariales tales como *clustering*, múltiples almacenes para mensajes, así como la capacidad de emplear cualquier administrador de base de datos como proveedor de persistencia *JMS*; *Apache Archiva* es una aplicación para administrar uno o más repositorios remotos, incluyendo administración, manipulación, navegación y búsqueda de artefactos, desarrollado por la *Apache Software Foundation*. Es capaz de interactuar con herramientas como *Maven*, *Continuum* y *Continuum Ant*, destinadas a producir una aplicación lista para usarse; *Apache Maven* es una herramienta de software para la gestión y construcción de proyectos *Java*, actualmente es uno de los principales proyecto de la *Apache Software Foundation*; *iText* es una herramienta para la generación de archivos *PDF* desde *Java*; *JFreeChart* es una librería de clases, escrita en *Java*, para generar gráficas por medio del uso de la *API* de *Java2D*, actualmente soporta gráficos de barras, gráficos circulares, gráficos de líneas, diagramas XY y gráficos de series temporales.

Herramienta	No. de proyectos	No. de clases	Tiempo de análisis*
iText Core	1	500	17 min
JFreeChart	1	500	12 min
Apache Maven	11	652	16 min
ActiveMQ	30	2,097	58 min
Apache Archiva	49	360	20 min
TOTAL	92	4,109	

Tabla 7 Tamaño de las herramientas evaluadas
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

Cabe hacer mención, que el tiempo de análisis (*) mostrado en la Tabla 7 se refiere al primer análisis satisfactorio de cada una de las herramienta analizadas, en posteriores análisis el tiempo puede variar ya que depende de la velocidad de la red en donde se ejecute, de la memoria RAM de la computadora en donde se procese, de la disponibilidad de los repositorios de código públicos en donde están almacenadas las herramientas evaluadas.

En la Tabla 8, se presentan las herramientas desglosadas en los proyectos que contiene y en el número de clases, vemos que existen dos herramientas que contienen sólo un proyecto, mientras que la herramienta que contiene más proyectos es *Apache Archiva* con 49 proyectos. Se procesaron en total 5 herramientas, 92 proyectos y 4,109 clases *Java*.

Herramienta	Proyecto	No. de clases	
iText Core	*iText Core*	500	500
JFreeChart	*JFreeChart*	500	500
Apache Maven	*Maven Plugin API*	19	652
	Maven Builder Support	7	
	Maven Model	1	
	Maven Model Builder	83	
	Maven Core	313	
	Maven Settings	1	
	Maven Settings Builder	27	
	Maven Artifact	27	
	Maven Aether Provider	14	
	Maven Embedder	29	
	Maven Compat	131	

Tabla 8 Herramientas evaluadas desglosadas por proyectos
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

Herramienta	Proyecto	No. de clases	
Apache Archiva	*Archiva :: Command Line Client*	3	360
	Archiva Base :: Test Utility	3	
	Archiva Base :: Common	9	
	Archiva Base :: Mocks	3	
	Archiva Base :: FileLock	5	
	Archiva Base :: Model	5	
	Archiva Base :: Configuration	17	
	Archiva Base :: Checksum	4	
	Archiva Base :: Plexus Bridge	4	
	Archiva Base :: Policies	17	
	Archiva Base :: Indexer	20	
	Archiva Base :: Consumers :: API	8	
	Archiva Base :: Consumers :: Core	15	
	Archiva Base :: Consumers :: Lucene	1	
	Archiva Base :: Consumers :: Metadata	1	
	Archiva Base :: Repository Interface Layer	14	
	Archiva Base :: XML Tools	6	
	Archiva Base :: Proxy Api	3	
	Archiva Base :: Proxy Common	5	
	Archiva Base :: Proxy	5	
	Archiva Base :: Transactions	6	
	Archiva Base :: Repository Converter	9	
	Archiva Base :: Repository Scanner	9	
	Archiva Base :: Repository Admin Default	14	
	Archiva Base :: Security Common	1	
	Archiva Base :: Maven 2 Metadata	1	
	Archiva Base :: Maven 2 Model	2	
	Archiva Scheduler :: API	2	
	Archiva Scheduler :: Indexing	8	
	Archiva Scheduler :: Repository Scanning Api	2	
	Archiva Scheduler :: Repository Scanning	3	
	Archiva Web :: Security Configuration	8	
	Archiva Web :: WebDAV	17	
	Archiva Web :: RSS	6	
	Archiva Web :: Tests Mocks	1	
	Archiva Web :: REST support :: Services	27	
	Archiva Web :: Web Common	18	
	Archiva Metadata :: Model	16	
	Archiva Metadata :: Repository API	3	
	Archiva Metadata :: Maven 2 Model	1	
	Archiva Core Plugins :: File System Backed Metadata Repository	2	
	Archiva Core Plugins :: Maven 2.x Repository Support	23	
	Archiva Core Plugins :: Repository Statistics	4	
	Archiva Core Plugins :: Problem Reporting Plugin	3	
	Archiva Core Plugins :: Audit Logging	6	
	Archiva Core Plugins :: Stage Repository Merge	3	
	Archiva Core Plugins :: Generic Metadata Support	2	
	Archiva Core Plugins :: JCR Storage for Metadata	3	
	Archiva Core Plugins :: Cassandra Storage for Metadata	12	

Tabla 8 Herramientas evaluadas desglosadas por proyectos (Continuación)
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

Herramienta	Proyecto	No. de clases	
ActiveMQ	*Openwire Generator*	18	2,097
	Client	500	
	Openwire Legacy Support	439	
	Broker	35	
	STOMP Protocol	24	
	MQTT Protocol	22	
	AMQP	65	
	KahaDB Store	67	
	JDBC Store	38	
	LevelDB Store	14	
	Unit Tests	500	
	Camel	26	
	Console	57	
	JAAS	15	
	Apache Karaf	2	
	Generic JMS Pool	17	
	Pool	5	
	ConnectionFactory	1	
	RA	26	
	Shiro	28	
	Spring	20	
	Runtime Configuration	26	
	Memory Usage Test Plugin	7	
	Performance Test Plugin	42	
	JUnit Rule	15	
	Web	43	
	Partition Management	6	
	OSGi bundle	1	
	Log4j Appender	3	
	HTTP Protocol Support	35	

Tabla 8 Herramientas evaluadas desglosadas por proyectos (Continuación)
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

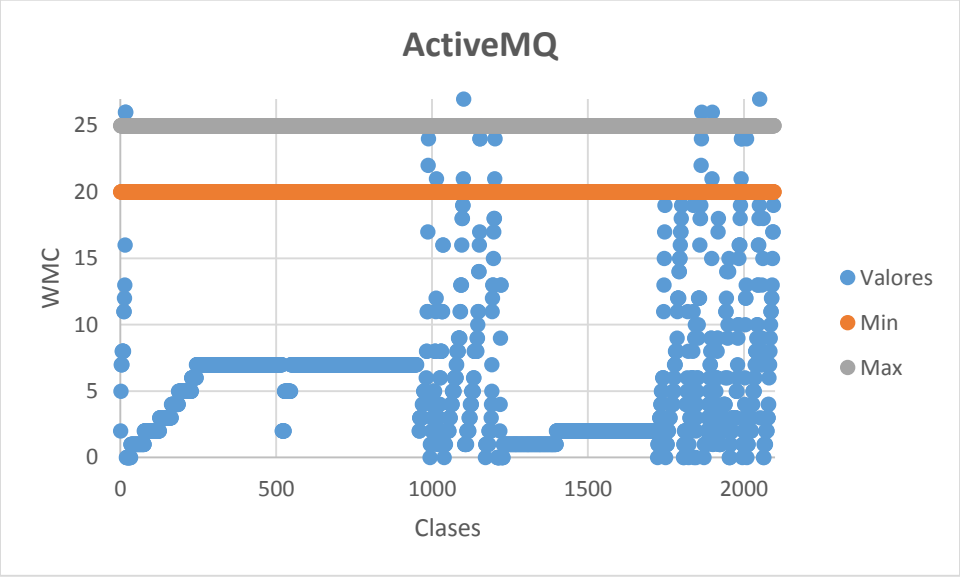
Tomando como base la información generada por la herramienta desarrollada y contenida en los archivos: *com.itextpdf-itextpdf.xls*, *org.apache.activemq-activemq-parent.xls*, *org.apache.archiva-archiva.xls*, *org.apache.maven-maven.xls* y *org.jfree-jfreechart.xls* se generaron gráficas para realizar el análisis de las métricas. Todos los tipos de archivos generados por la herramienta desarrollada debido a la gran cantidad de información contenida y su tamaño, no formarán parte de este documento y estarán disponibles en el disco compacto (CD) que será entregado junto con ésta investigación.

Análisis de las métricas CK

Las métricas creadas por Shyam R. Chidamber y Chris F. Kemerer son utilizadas para medir la calidad en el desarrollo de aplicaciones orientadas a objetos, estas métricas no están enfocadas a la productividad, ya que tratan de evaluar de manera cuantitativa, si ciertas propiedades deseables del diseño orientado a objetos se cumplen. A continuación se presenta el resultado del análisis realizado a 92 proyectos.

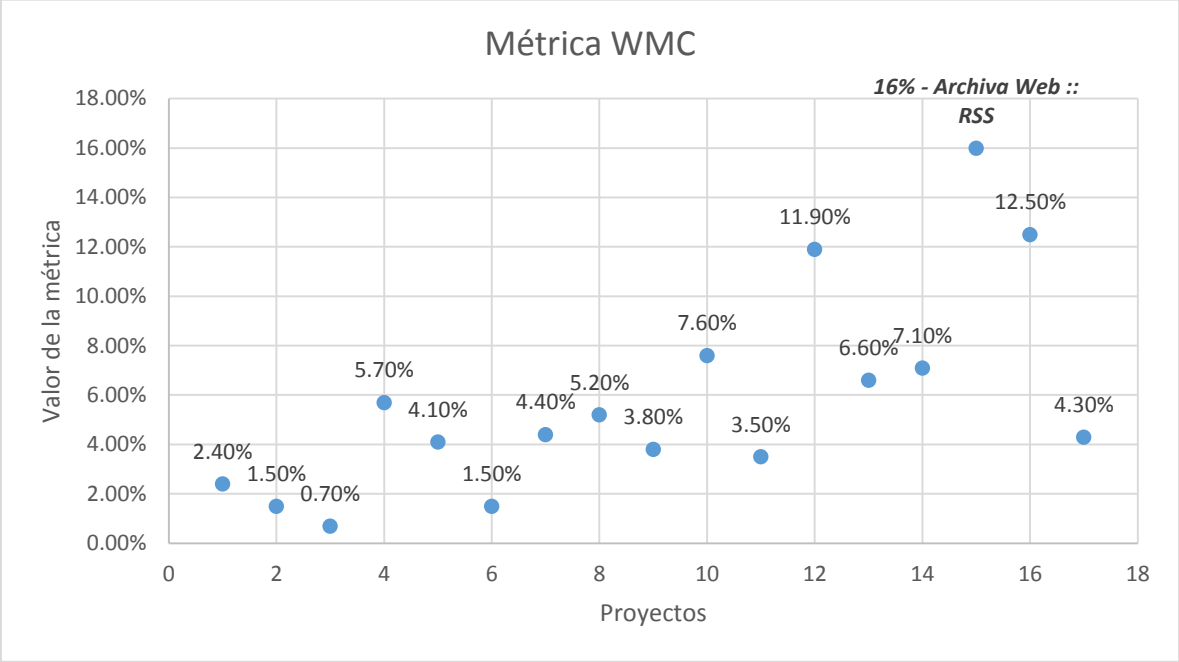
5.1.1 Métrica WMC

De los 92 proyectos, únicamente 17 proyectos cumplen con el rango óptimo que es de 20 a 25 (Misra & Bhavsar 2003), esto es el 18.48%; siendo la herramienta *ActiveMQ* la que 13 de sus proyectos se situaron en el rango óptimo, como lo vemos en la Gráfica 1.



Gráfica 1 Valores de WMC
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

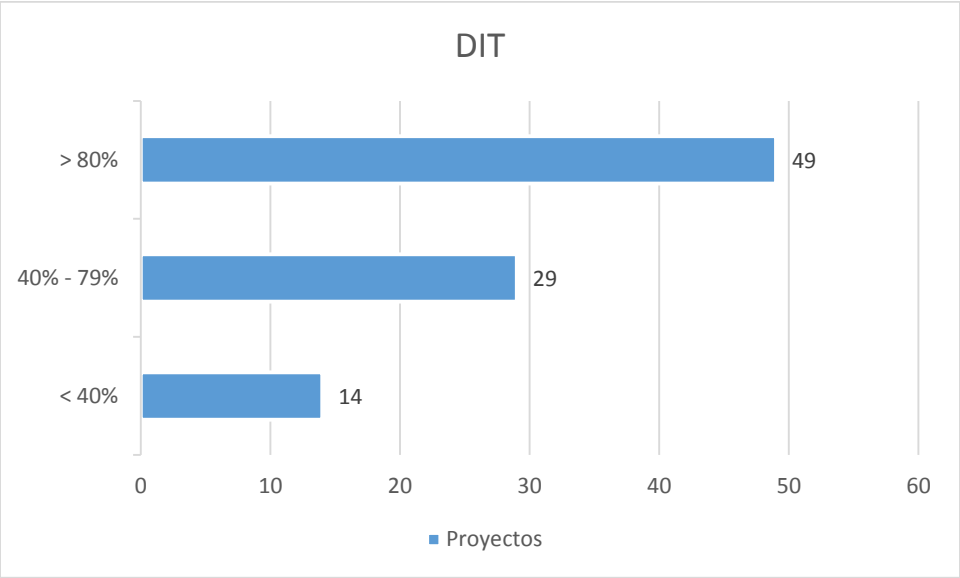
En la Gráfica 2, podemos observar que el proyecto "*Archiva Web :: RSS*" fue el proyecto que alcanzó el mayor valor con un 16%; la métrica WMC mide la complejidad de una clase y si se obtuvieron valores muy bajos, eso implica que las clases analizadas no son complejas, con esto disminuye el tiempo para su mantenimiento y aumenta la posibilidad de ser reutilizadas.



Gráfica 2 Resultado WMC
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

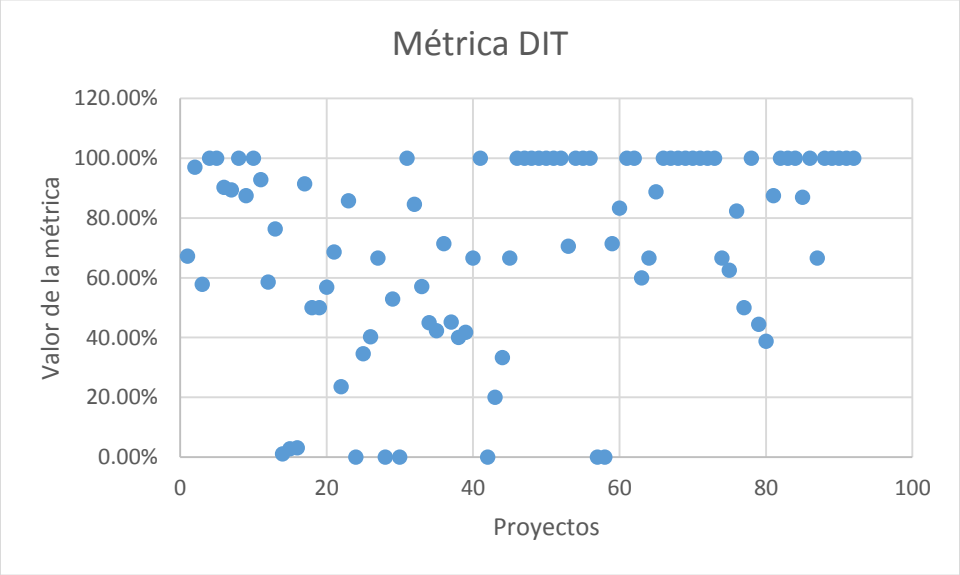
5.1.2 Métrica DIT

En la Gráfica 3 vemos como están distribuidos los valores obtenidos de la métrica DIT, cuyo rango óptimo es de 1 a 5 (Misra & Bhavsar 2003); en la gráfica podemos identificar que más del 50% de las clases analizadas, cumplen en un 80% con el rango óptimo y el 15.22% tiene un valor por debajo del 40%, por lo que podemos concluir que la mayoría de las clases analizadas cumplen con la métrica.



Gráfica 3 Métrica DIT
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

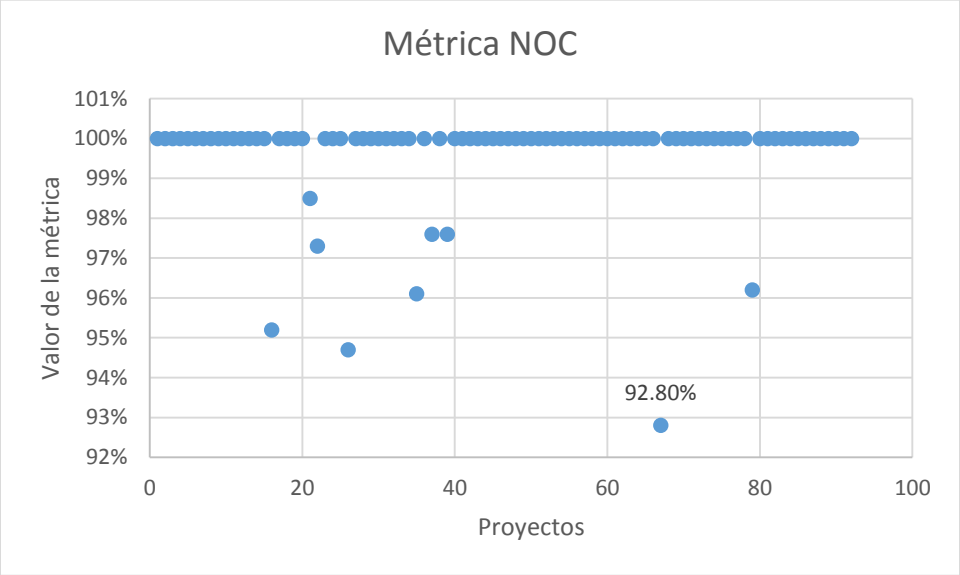
La métrica DIT mide los niveles de las clases en la jerarquía de herencia, el 80% de las clases analizadas al cumplir con esta métrica (ver Gráfica 4), indica que presentan una complejidad media y que tienen grandes posibilidad de ser reutilizadas.



Gráfica 4 Resultados DIT
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

5.1.3 Métrica NOC

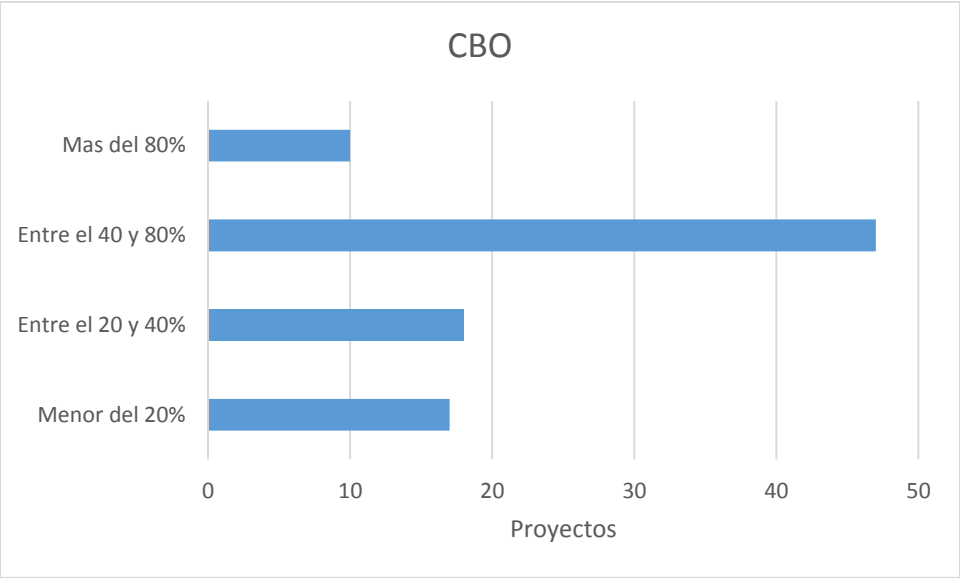
La métrica NOC es el número de subclases que pertenecen a una clase, el valor óptimo es menor a 5 (Misra & Bhavsar 2003); en la Gráfica 5 vemos la distribución de la métrica NOC y podemos comprobar que los valores que presentan las clases analizadas son por encima del 90%; este resultado es un indicador del nivel de reúso de las clases.



Gráfica 5 Resultados NOC
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

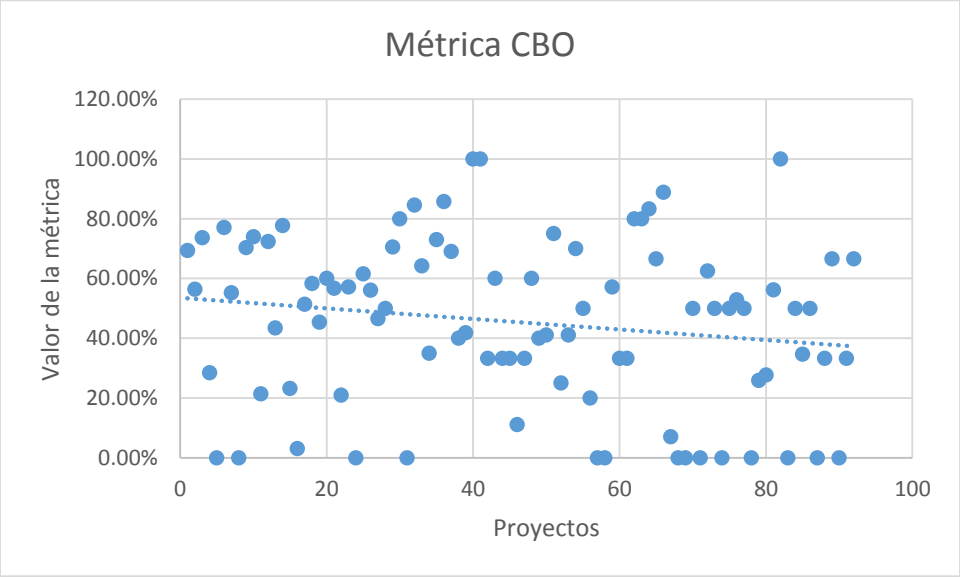
5.1.4 Métrica CBO

En la Gráfica 6 vemos como se encuentran distribuidos los valores obtenidos de la métrica CBO, cuyo rango óptimo es de 1 a 4 (Misra & Bhavsar 2003); más del 50% de proyectos analizados, cumplen con el rango óptimo de la métrica entre un 40 y un 80%; esta métrica indica el acoplamiento entre objetos.



Gráfica 6 Métrica CBO
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

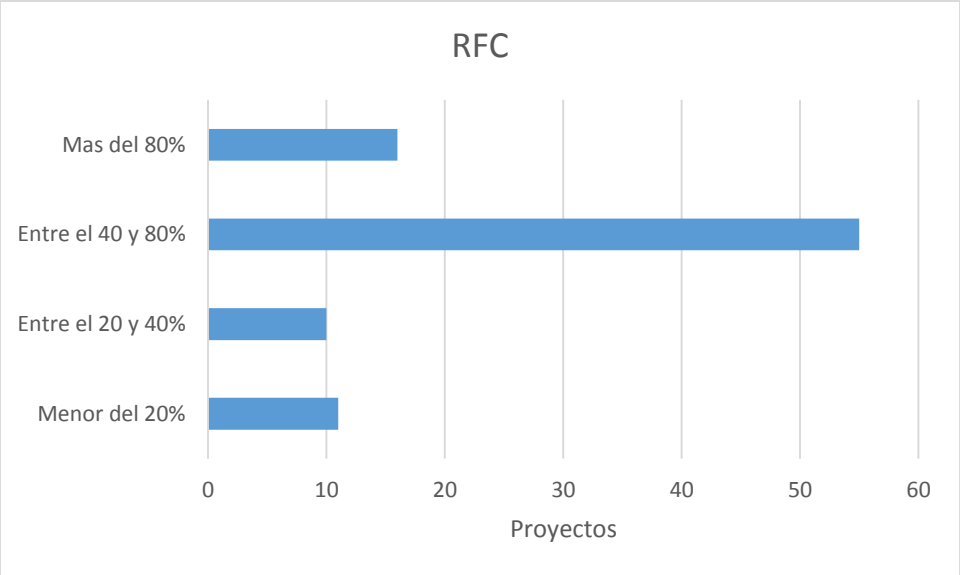
La distribución total de los resultados de la métrica, se presentan en la Gráfica 7, en donde se puede identificar, que la tendencia de la métrica CBO va a la baja, esto es importante ya que el acoplamiento indica el nivel de dependencia de objetos y el bajo acoplamiento entre objetos es el estado ideal que siempre se desea al momento de diseñar programas de cómputo.



Gráfica 7 Resultados CBO
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

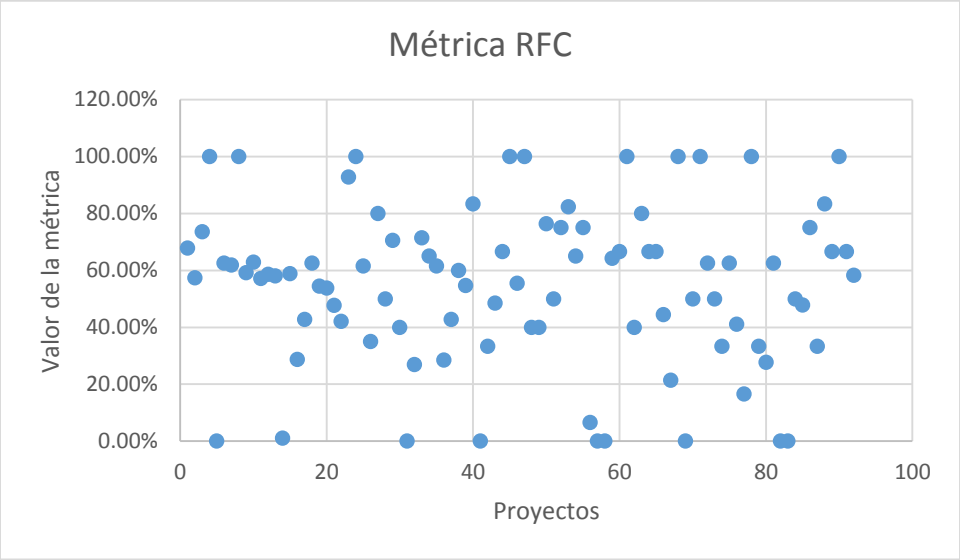
5.1.5 Métrica RFC

La métrica RFC es una medida de la complejidad de una clase a través del número de métodos y de la comunicación con otras clases, en la Gráfica 8, vemos que más del 50% de proyectos analizados cubren con el valor óptimo de la métrica, el cual es de 15 o menor (Misra & Bhavsar 2003).



Gráfica 8 Métrica RFC
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

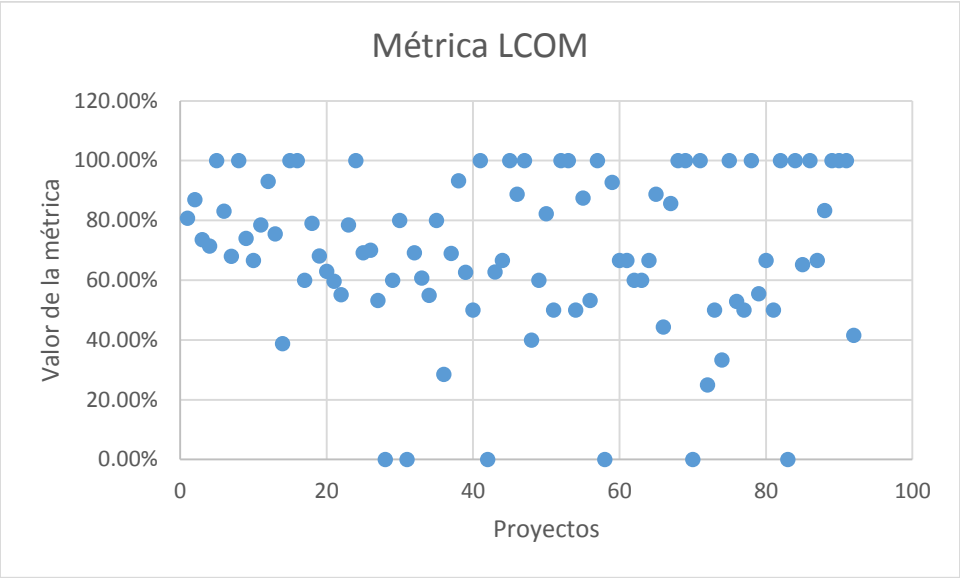
La distribución de la métrica RFC obtenida al analizar los 92 proyectos, es mostrada en la Gráfica 9; mientras más grande sea el valor de la métrica RFC de una clase, hace que la clase sea más compleja y más difícil sea su mantenimiento.



Gráfica 9 Resultados RFC
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

5.1.6 Métrica LCOM

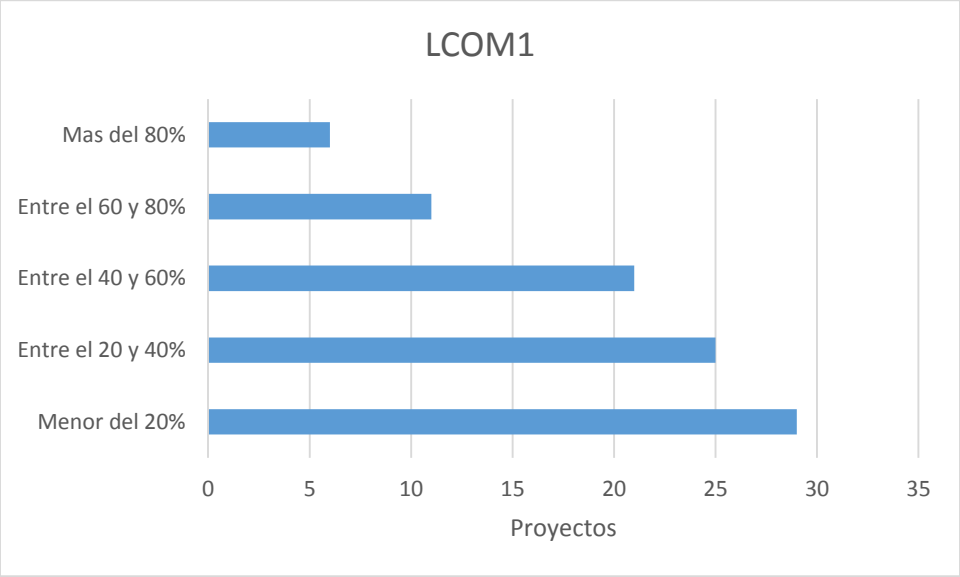
La métrica LCOM es una medida de la cohesión de una clase teniendo en cuenta el número de atributos comunes usados por diferentes métodos; LCOM indica la calidad de la abstracción hecha en la clase; en la Gráfica 10, vemos la distribución de los valores obtenidos en el análisis, en donde más del 70% de los proyectos analizados cumplen por arriba del 60% con el valor óptimo para ésta métrica, que es de 0 (Misra & Bhavsar 2003); la importancia de LCOM radica en que al tener un valor alto implica falta de cohesión, incrementando la complejidad y la probabilidad de errores durante el desarrollo, en cambio un valor bajo indica una alta cohesión en los métodos dentro de una clase lo que implica que ésta no pueda ser dividida, lo cual fomenta la encapsulación.



Gráfica 10 Resultados LCOM
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

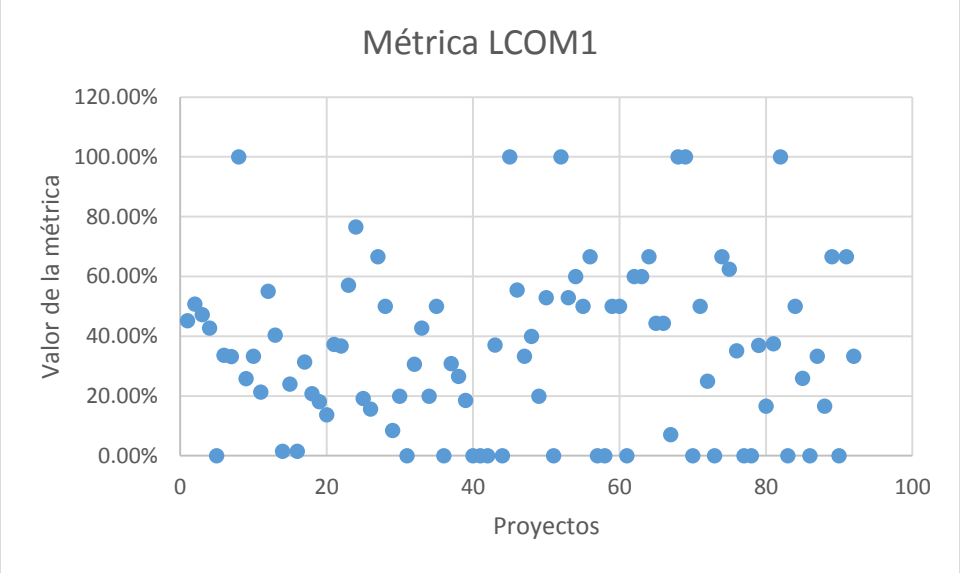
5.1.7 Métrica LCOM1

La métrica LCOM1, es una variante de la métrica LCOM propuesta por Li y Henry (Olmedilla 2005) en donde modificaron la fórmula para realizar el cálculo de la métrica; en la Gráfica 11 vemos un gran contraste en relación con la métrica LCOM original, ya que menos del 40% de los proyectos cumplieron con el valor óptimo de la métrica.



Gráfica 11 Métrica LCOM1
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

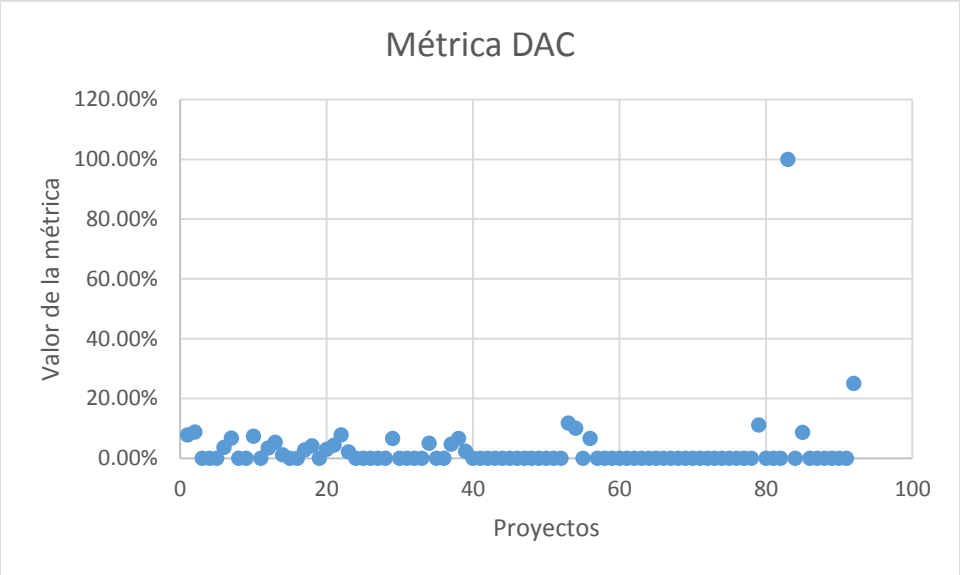
La Gráfica 12, muestra la distribución total de los valores obtenidos de la métrica LCOM1; la modificación propuesta por Li y Henry para el cálculo de la métrica LCOM1, implican la exclusión de la herencia y de los constructores de las clases.



Gráfica 12 Resultados LCOM1
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

5.1.8 Métrica DAC

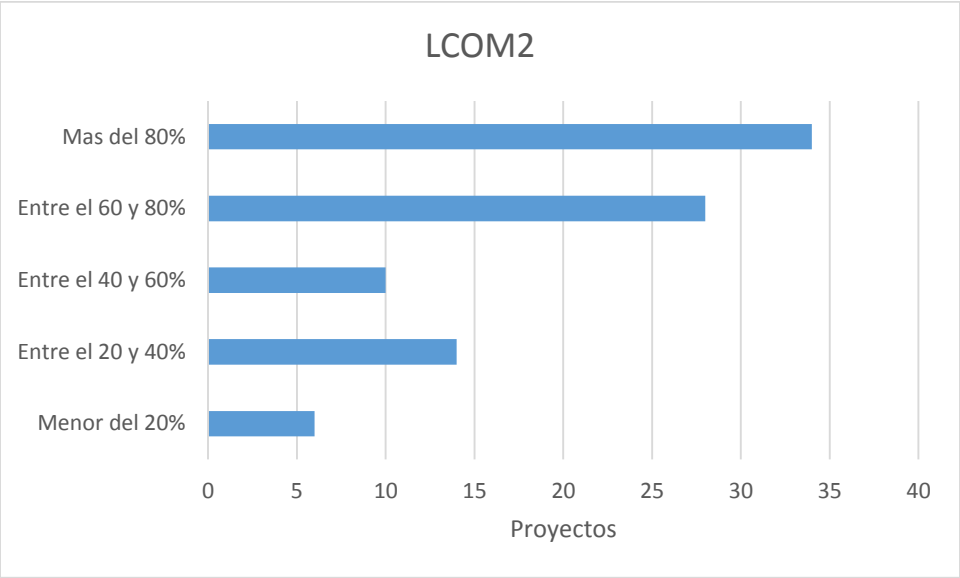
La métrica DAC permite obtener el número de atributos de una clase que tienen como tipo a otra clase, generando un tipo particular de acoplamiento que puede violar la propiedad de encapsulación. En la Gráfica 13, vemos la distribución de los valores obtenidos al analizar los 92 proyectos, y claramente se puede observar que la gran mayoría de proyectos obtuvo un valor de 0, lo cual nos indica que no existe o es muy bajo el nivel de acoplamiento complejo causado por la abstracción de datos.



Gráfica 13 Resultados DAC
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

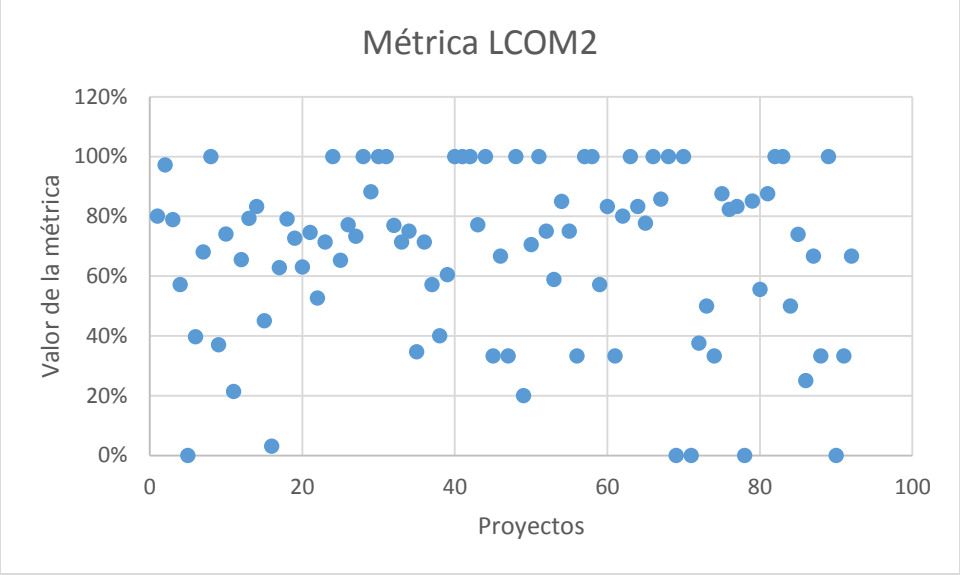
5.1.9 Métrica LCOM2

Al igual que la métrica LCOM1, la métrica LCOM2 es una variación propuesta por Henderson-Sellers; la modificación que agregaron, la definen como una normalización del número de métodos y variables de una clase (Badri et al. 2011). En la Gráfica 14, se puede apreciar a diferencia de la métrica LCOM1, que la mayoría de proyectos analizados cumplen con el rango óptimo para la métrica, que es entre 0 y 1 (Olmedilla 2005).



Gráfica 14 Métrica LCOM2
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

La gráfica que a continuación se muestra (Gráfica 15), se observa la distribución total de los valores obtenidos de la métrica LCOM2. Un valor bajo de LCOM2 indica alta cohesión, una clase bien diseñada, una buena subdivisión de clase que implique simplicidad y alta reutilización; mientras que un valor alto de LCOM2 indica disminución de la encapsulación y aumento de la complejidad, aumentando así la probabilidad de errores.



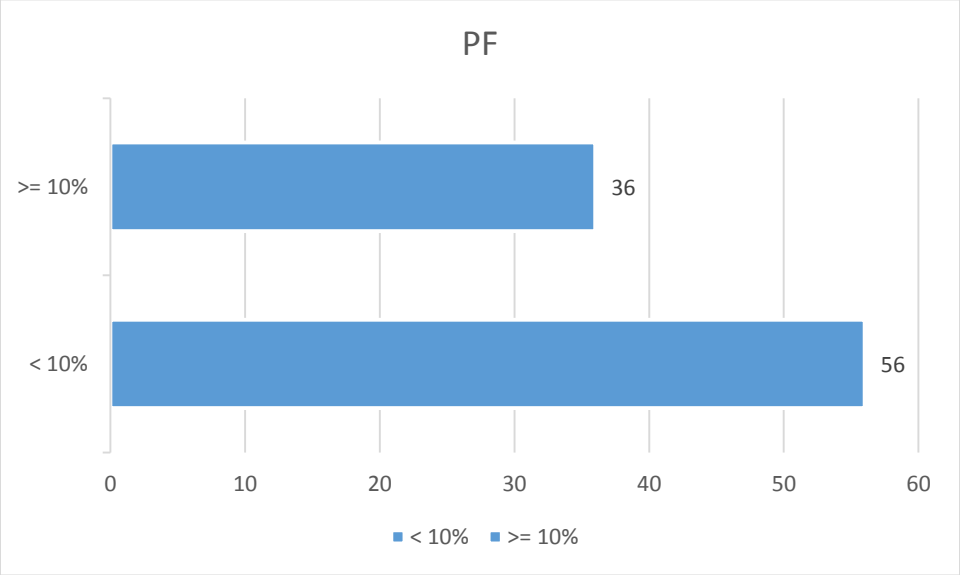
Gráfica 15 Resultados LCOM2
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

5.2 Análisis de las métricas MOOD

Estas métricas (a diferencia de las métricas CK que realizan la evaluación a nivel de clase), consideran distintos niveles de granularidad, realizan la evaluación de todo el sistema de acuerdo a la herencia, acoplamiento, polimorfismo y encapsulamiento, ésta diferencia hace que éstas métricas están enfocadas a la productividad. A continuación se presentan los resultados obtenidos al realizar el análisis de las métricas en 92 proyectos.

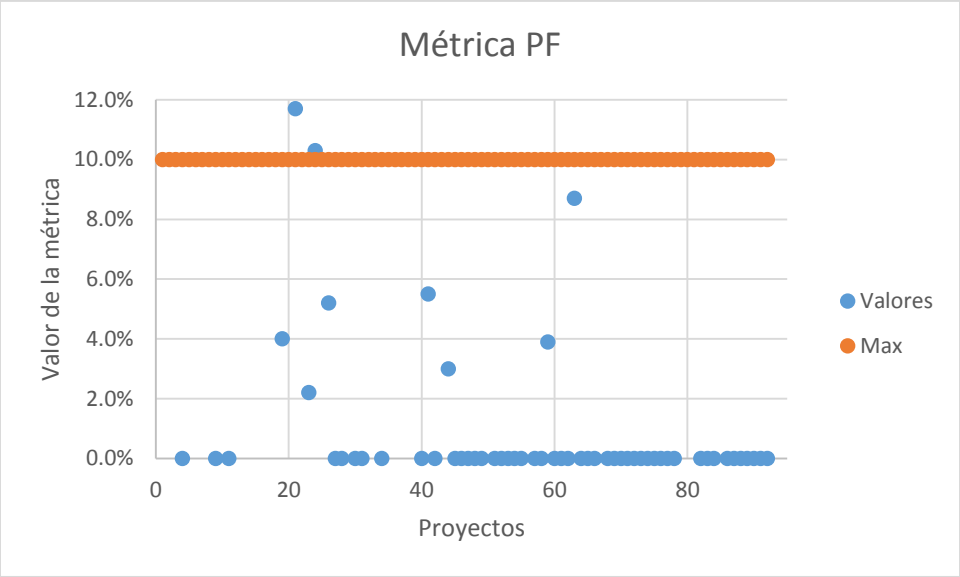
5.2.1 Métrica PF

Como se puede constatar en la Gráfica 16, más del 50% de los 92 proyectos analizados, cumple con el valor óptimo de la métrica PF, la cual es menor al 10%; esta métrica representa el número de posibles situaciones polimórficas diferentes, es además, una medida indirecta de la asociación dinámica del sistema.



Gráfica 16 Métrica PF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

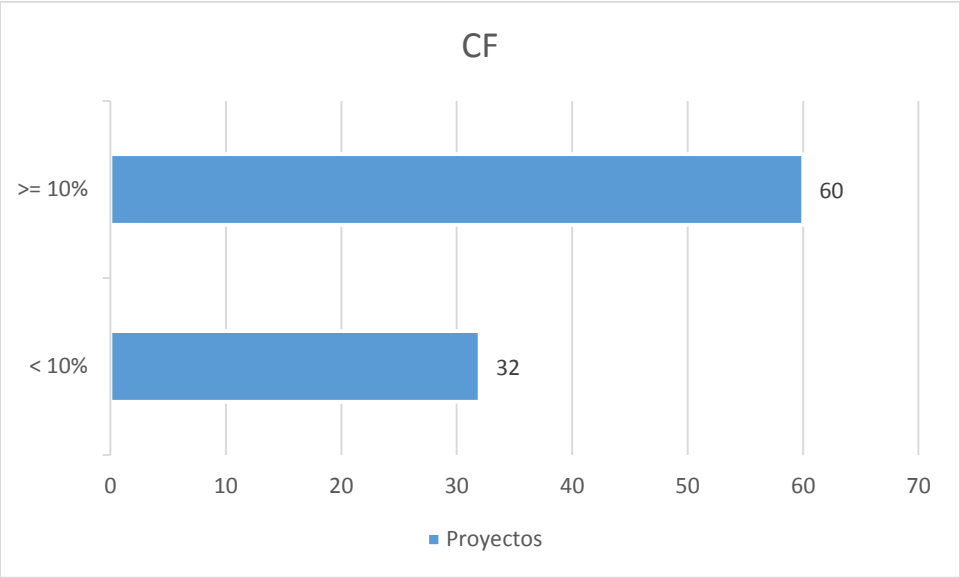
En la Gráfica 17 se presenta la distribución de los valores obtenidos de la métrica PF, como se puede apreciar, la gran mayoría de valores son 0%; como el polimorfismo es debido a la herencia, el cumplir con esta métrica significa que la sobrecarga de los métodos heredados reduce la complejidad, se incrementa el mantenimiento y hace que el sistema sea más fácil de entender.



Gráfica 17 Resultados PF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

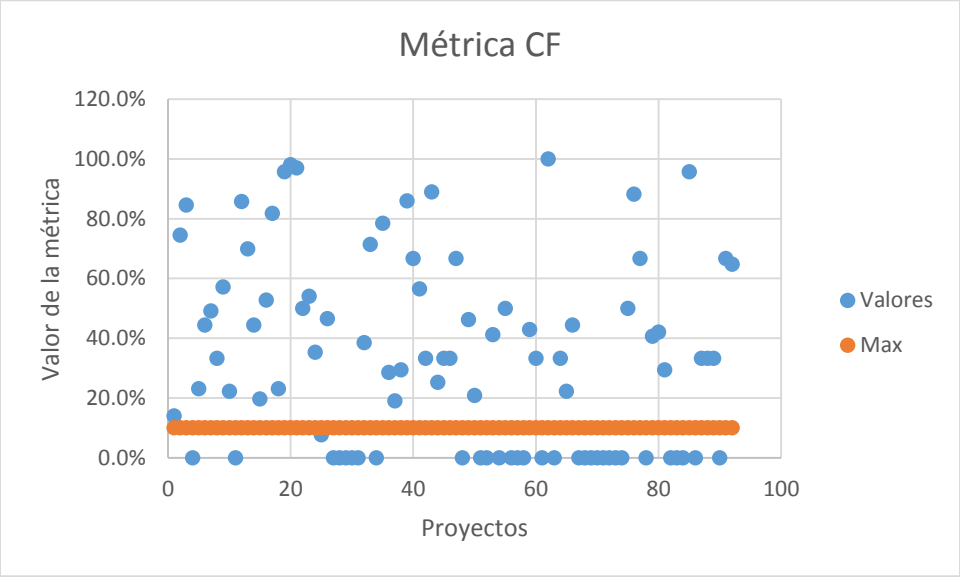
5.2.2 Métrica CF

En la Gráfica 18, se observa que 60 de los 92 proyectos, superan el valor óptimo de la métrica CF; dicha métrica mide la proporción entre el número máximo posible de acoplamientos del sistema y el número real de acoplamientos no imputables a herencia; estos resultados sugieren un cambio incrementando la encapsulación ya que existe un alto acoplamiento entre las clases.



Gráfica 18 Métrica CF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

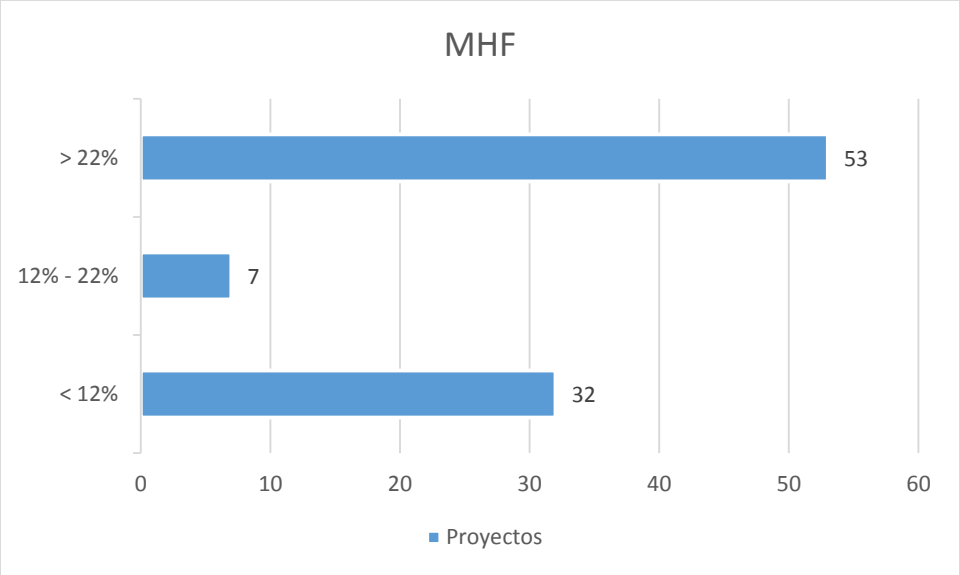
En la Gráfica 19, se muestra la distribución de los 92 proyectos. Podemos observar que la gran mayoría de proyectos superan el 10%, que es el valor óptimo; el tener un alto acoplamiento puede ocasionar un incremento en la densidad de defectos generando una gran dificultad para su mantenimiento.



Gráfica 19 Resultados CF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

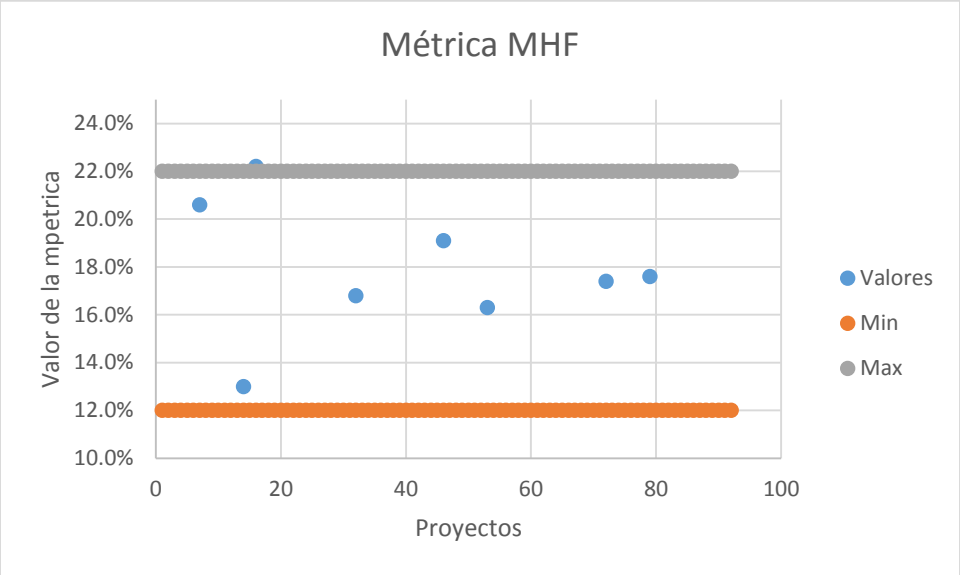
5.2.3 Métrica MHF

La métrica MHF es la proporción entre los métodos definidos como protegidos o privados y el número total de métodos. En la Gráfica 20, se presentan los proyectos analizados. Los resultados obtenidos sugieren incrementar la encapsulación de la información de las clases.



Gráfica 20 Métrica MHF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

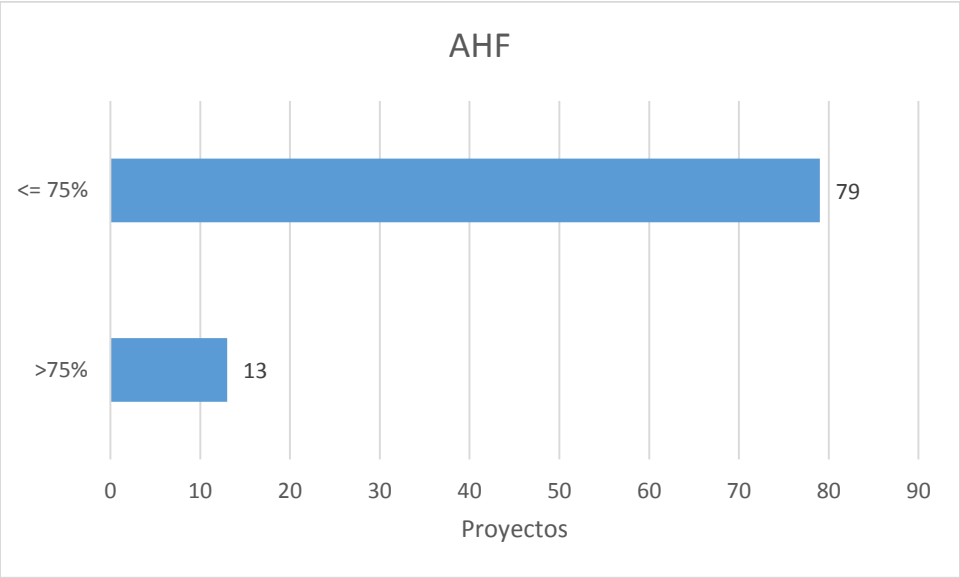
En la Gráfica 21, se presentan los 7 de los 92 proyectos, que se encuentran en el rango óptimo que es del 12 al 22%. Ya que menos del 10% de los proyectos analizados cumplen con la métrica MHF, ocasiona que se generen una gran densidad de defectos, dificultando su mantenimiento y el esfuerzo necesario para corregirlos sea grande.



Gráfica 21 Resultados MHF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

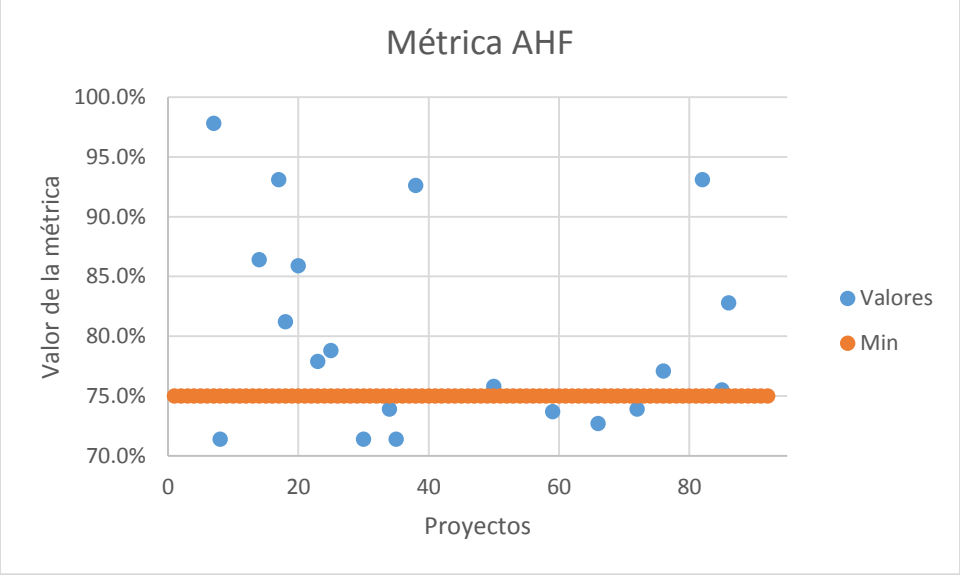
5.2.4 Métrica AHF

La métrica AHF es la proporción entre los atributos definidos como protegidos o privados y el número total de atributos. En la Gráfica 22, vemos que únicamente 13 de los 92 proyectos analizados cumplen con el valor óptimo que es del 75%. Idealmente esta métrica debe de ser siempre 100%, intentando ocultar todos los atributos. Se sugiere que no se deben usar atributos públicos, ya que se considera que violan los principios de encapsulación al exponer la implementación de las clases.



Gráfica 22 Métrica AHF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

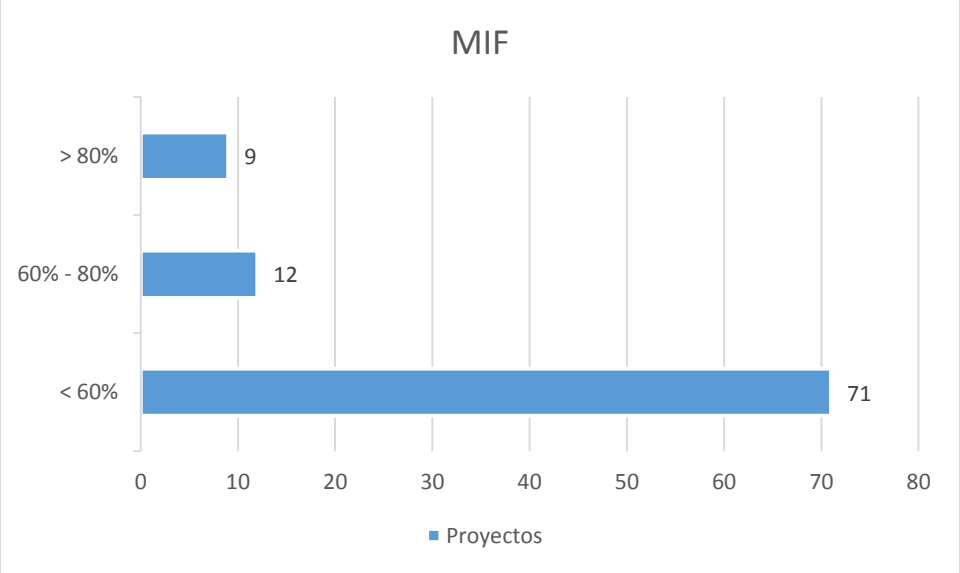
En la Gráfica 23, se muestra la distribución de los 92 proyectos analizados, que obtuvieron un valor por arriba del 70%. Se puede observar que más del 80% de los proyecto no cumplen con la métrica AHF, por lo que se debe de realizar una reingeniería de los proyectos para aplicar la encapsulación, ya que es una mala práctica acceder a los atributos directamente.



Gráfica 23 Resultados AHF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

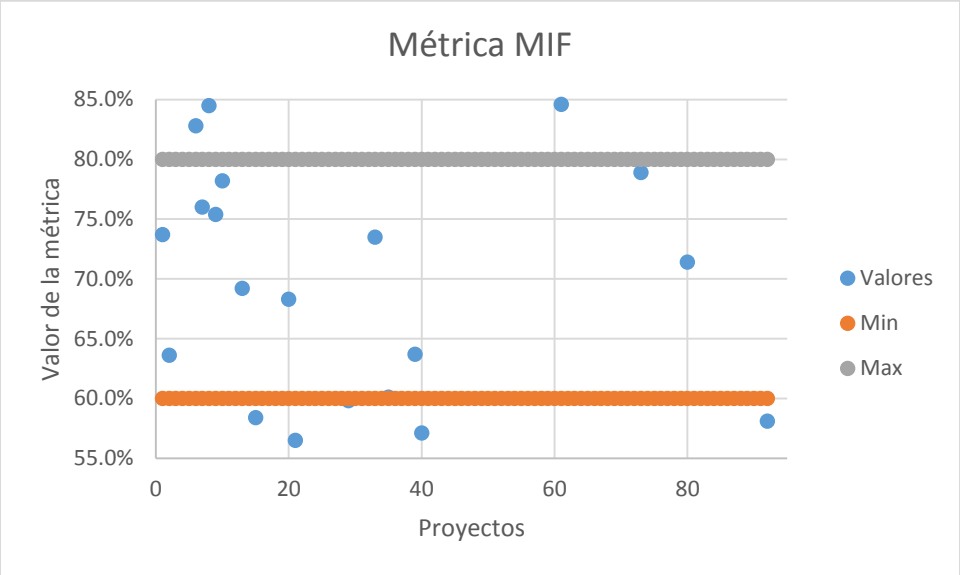
5.2.5 Métrica MIF

La métrica MIF es la proporción de la suma de todos los métodos heredados de todas las clases entre el número total de métodos (localmente definidos más los heredados) en todas las clases que componen un sistema. El rango óptimo es del 60 al 80%; en la Gráfica 24 se ve claramente que únicamente 12 de los 92 proyectos analizados cumplen con el valor óptimo. La falta de cumplimiento de la métrica MIF sugiere replantear el diseño de los proyectos para incrementar el uso de la herencia y de la sobre escritura de los métodos heredados.



Gráfica 24 Métrica MIF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

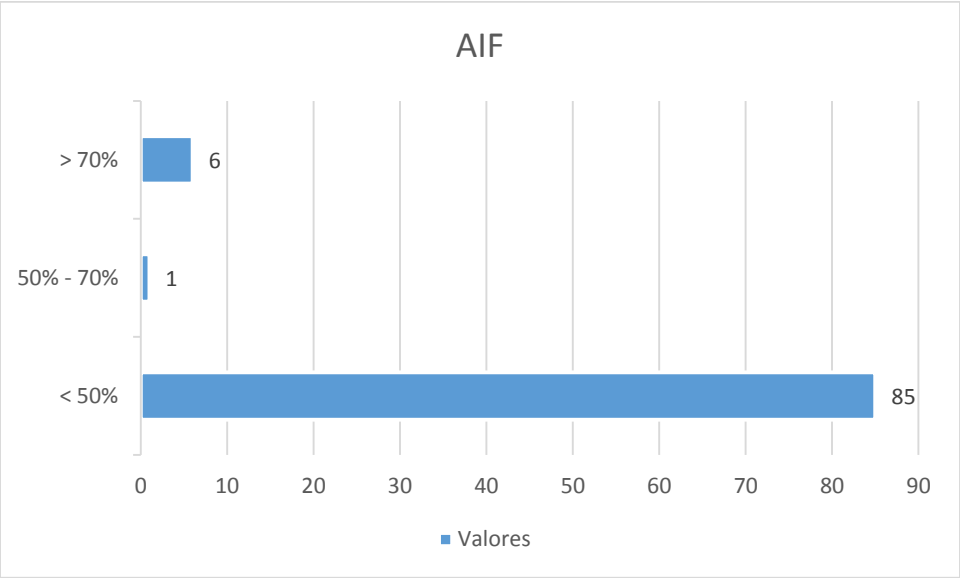
En la Gráfica 25 se presentan los 12 proyectos que se encuentran en el rango óptimo para la métrica MIF. El incremento en el uso de la herencia en las clases tiene un efecto directo sobre la reusabilidad, la comprensión y el mantenimiento del sistema.



Gráfica 25 Resultados MIF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

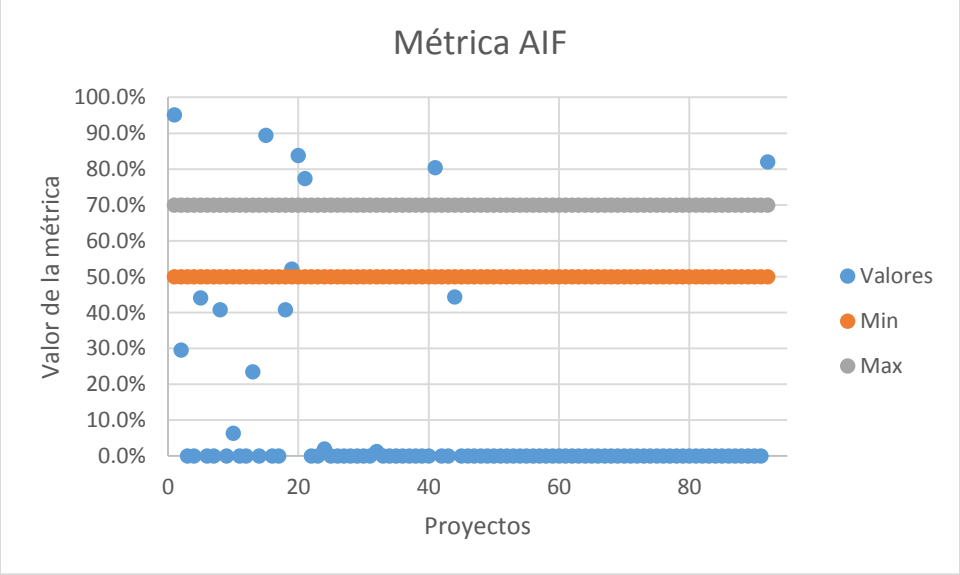
5.2.6 Métrica AIF

La métrica AIF se define como la proporción del número de atributos heredados entre el número total de atributos. En la Gráfica 26 se observa que solamente 1 proyecto de los 92 proyectos analizados cumple con el rango óptimo que es del 50 al 70%. Los resultados obtenidos para ésta métrica sugieren el uso de atributos heredados, ya que la gran mayoría de los proyectos no hereda atributos, ocasionando un baja reusabilidad.



Gráfica 26 Métrica AIF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

En la Gráfica 27, se presenta la distribución total de los valores obtenidos por la métrica AIF. Se observa claramente, que la gran mayoría de los proyectos no hacen uso de la herencia de los atributos, dificultando el reuso de las clases y su futuro mantenimiento.



Gráfica 27 Resultados AIF
Elaborada por Romero Ricardo, (2016) con datos analizados en esta investigación

6.- CONCLUSIONES Y TRABAJO FUTURO

En esta investigación se generó una herramienta para el análisis de la calidad del software en proyectos de software diseñados con el paradigma orientado a objetos, por medio de la automatización de la verificación de métricas de código en un ambiente de integración continua.

Aunque en un principio, la idea era construir dicha herramienta como un *plugin* del servidor de integración continua (*Jenkins*), se terminó creando un *plugin* de la herramienta de control de calidad de código (*SonarQube*), ya que ésta herramienta fue desarrollada para que la gestión de la calidad del código sea accesible para todo el mundo con un mínimo esfuerzo, ofrece analizadores de código, herramientas de informes, módulos de detección de defectos y porque obtiene diversas métricas básicas entre las cuales están: las métricas de comentarios, de reglas de codificación, de código duplicado, de pruebas unitarias y de la complejidad del código.

Con el desarrollo de la presente investigación, se logró comprobar la hipótesis planteada, en la cual se consideró en un marco de trabajo de los servidores de IC, la posibilidad que generar la funcionalidad que permita recolectar información de proyectos de software orientados a objetos, para obtener las métricas CK y MOOD de forma automatizada.

Para probar la eficacia de la herramienta desarrollada, se analizaron cinco herramientas, las cuales están integradas por 92 proyectos, cuyas características son: pertenecer a la categoría de software libre, creadas con lenguaje *Java* y la herramienta *Maven*, así como contar con una extensión de más de 50,000 líneas de código.

El resultado de la aplicación de las métricas CK y MOOD a las 5 herramientas fue muy satisfactorio debido a que se logró: i) obtener las métricas en forma distribuida por medio del análisis en paralelo de las herramientas, en un tiempo relativamente corto; ii) exportar los valores obtenidos en los 3 formatos previstos (*PDF*, *XLS* y *CSV*) y iii) crear el componente como un *plugin* para extender la funcionalidad de la herramienta de control de calidad de código (*SonarQube*).

La herramienta desarrollada en esta investigación es un buen punto de partida para elaborar en trabajos futuros, una herramienta más robusta, eficiente y funcional para el análisis de métricas de calidad de software.

El trabajo se puede orientar y fortalecer en las siguientes temáticas:

Mejoras a la herramienta: Agregar funcionalidad para la generación automática de gráficas con los datos obtenidos mediante el análisis del componente desarrollado; implementar la configuración de los valores óptimos de cada una de las métricas CK y MOOD; incorporar nuevas métricas que sean del interés de otros grupos de trabajo; modificar o replantear las fórmulas de las métricas CK y MOOD que en este trabajo de investigación se generaron; por último desarrollar nuevos formatos de archivos de salida para facilitar la manipulación de los valores obtenidos por el componente desarrollado.

Aplicación de la herramienta: Que permitiría analizar y contrastar proyectos de software creados con diferentes metodologías de desarrollo; que permita contrastar los diferentes niveles de experiencia de grupos de desarrolladores, mediante el análisis de proyectos creados por dichos grupos; por último, tener una referencia de las tendencias de implementación de software en diversos países, mediante el análisis de proyectos creados por 2 o más empresas de desarrollo de software de diferentes países.

APENDICES

APÉNDICE 1. ENTREGABLES FASE INICIO

Los documentos que componen esta sección son:

Visión del proyecto

Plan del proyecto

Requerimientos del proyecto

Visión del proyecto

MetricasCKMOOD
Documento de Visión

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Visión del Proyecto.pdf	0.1	01/12/2015	Ing. Ricardo Romero Villagómez	Borrador
Visión del Proyecto v1.0.pdf	1.0	07/12/2015	Ing. Ricardo Romero Villagómez	Primera versión
Visión del Proyecto v2.0.pdf	2.0	15/12/2015	Ing. Ricardo Romero Villagómez	Segunda versión
Visión del Proyecto v3.0.pdf	3.0	07/02/2016	Ing. Ricardo Romero Villagómez	Documento final

Contenido

1. Introducción

La calidad del software está estrechamente vinculada con la medición del mismo. La medición de la calidad del software es una necesidad para las empresas de desarrollo de software ya que representa una ventaja estratégica al proporcionar el conocimiento de los procesos productivos y permitir mejoras en las tareas menos eficientes, ya que siempre habrá elementos cualitativos para la creación de software.

Las métricas de software tienen un papel decisivo en la obtención de un producto de alta calidad, porque determinan mediante estadísticas basadas en la experiencia, el avance del software y el cumplimiento de parámetros requeridos.

Este proyecto presenta una propuesta orientada a contribuir a la calidad de proyectos de software diseñados con el paradigma orientado a objetos, automatizando el proceso de verificación de métricas de código, con el objetivo de evaluar, de forma cuantitativa, si ciertas propiedades deseables del diseño orientado a objetos son cumplidas.

2. Posicionamiento

2.1 Descripción del Problema

Problema	En los últimos años hemos visto un incremento considerable del uso del paradigma orientado a objetos en el desarrollo de software. El uso de técnicas de desarrollo de software orientado a objetos introduce nuevos elementos a la complejidad del software, tanto en el proceso de desarrollo de software como en el producto final. Muchas métricas orientadas a objetos se han propuesto específicamente con el fin de evaluar el diseño de un sistema de software. Sin embargo, la mayoría de los enfoques existentes para la medición de estos parámetros de diseño implican sólo algunos de los aspectos del paradigma orientado a objetos. Las métricas aquí planteadas (CK y MOOD) tratan de captar los diferentes aspectos del software como producto y su proceso de desarrollo. Estas métricas se pueden aplicar en varios proyectos para evaluar y comparar el rendimiento del código utilizando el paradigma orientado a objetos.
Afecta	Arquitectos de software e Ingenieros de software
Impacto	Extraer las métricas CK y MOOD en proyectos creados en lenguaje Java y que usen el paradigma orientado a objetos, pueden ser usados como referencia para la toma de decisiones, para la reducción de costos y para la mitigación de riesgos en empresas de desarrollo de software.
Una solución exitosa sería	Obtener las métricas CK y MOOD de forma automatizada Las métricas se presentan en una página web Las métricas obtenidas se pueden exportar en diferentes formatos (CSV, PDF, XLS)

2.2 Posicionamiento del Producto

Dirigido a	Arquitectos de software e Ingenieros de software
Las Métricas CKMOOD	Es una herramienta de control de calidad de software
Beneficios	Detecta de forma temprana problemas en el diseño del software Ayudará en la toma de decisiones sobre la re ingeniería del diseño Reducirá costos en la generación de software ya que no tendrá costo Permitirá mitigar riesgos
Alternativas competitivas	*Kiuwan Software Analytics* *HP Fortify Source Code Analyzer* *IBM Rational AppScan Source Edition* *Team Foundation Server* *CAST Application Intelligence Platform*
Características del producto	El software generado será de la categoría de software libre, por lo que se acogerá a las reglas de las licencias generales públicas (GNU GPL v3.0) La generación de las métricas se realizará de forma automática y se podrá calendarizar Se podrán obtener las métricas de forma distribuida

3. Partes Interesadas

3.1 Resumen de Partes Interesadas

Nombre	Descripción	Responsabilidades
Arquitectos de software	Persona clave que ayudará a tomar decisiones estratégicas para aprovechar al máximo la tecnología en el desarrollo de software.	Definir los lineamientos de diseño Definir la arquitectura Tomar decisiones sobre cuestiones técnicas Coordinar al equipo de desarrollo
Ingenieros de software	Persona que investiga, diseña y desarrolla sistemas de software	Investigar, diseñar y escribir nuevos programas con lenguajes de programación Crear especificaciones técnicas y comprobar los planes para mostrar cómo sus proyectos cubrirán las necesidades del proyecto Probar los nuevos programas y encontrar sus fallos Investigar nuevas tecnologías Redactar documentación técnica Monitorear y corregir los defectos del software

4. Descripción del Producto

4.1 Necesidades y Características

Necesidad	Prioridad	Característica	Versión planeada
Obtener métricas de diseño de software en proyectos de software diseñados con el paradigma orientado a objetos	Alta	Obtener las métricas de CK (WMC, DIT, NOC, CBO, RFC, LCOM, LCOM1, LCOM2, MPC, DAC, SIZE2) y MOOD (PF, CRF, MHF, AHF, MIF, AIF)	1.0
Obtener las métricas CK y MOOD de forma automática	Alta	Vincular el componente con un servidor de integración continua para obtener las métricas de forma automática	1.0
Presentar las métricas obtenidas en una página web	Alta	Presentar los resultados de las métricas obtenidas en un navegador web de forma clara	1.0
Trabajar en un ambiente distribuido	Alta	Instalar el componente en más de un nodo de una red para ser configurado y ejecutado desde el servidor de integración continua	1.0
Exportar las métricas obtenidas en diferentes formatos	Media	Exportar los resultados de las métricas en los formatos CSV, PDF, XLS	2.0

5. Otros Requerimientos del Producto

Requerimiento	Prioridad	Versión planeada
La aplicación debe de ser tolerante a fallos, en caso de error en la aplicación debe de mostrar una pagina que explique la causa del error	Alta	1.0
La configuración y uso del componente debe de ser intuitivo y lo más simple posible	Alta	1.0
El código debe de estar documentado	Alta	1.0
El componente por lo menos debe de funcionar en los sistemas operativos windows 7 y linux 7 distribución centOS	Alta	1.0
La aplicación debe de contar con un manual de instalación, de configuración y de uso	Alta	1.0
El código fuente a analizar deber de ser en el lenguaje Java	Alta	1.0

Plan del proyecto

MetricasCKMOOD
Plan del Proyecto

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Plan del Proyecto.pdf	0.1	01/12/2015	Ing. Ricardo Romero Villagómez	Borrador
Plan del Proyecto v1.0.pdf	1.0	07/12/2015	Ing. Ricardo Romero Villagómez	Primera versión
Plan del Proyecto v2.0.pdf	2.0	15/12/2015	Ing. Ricardo Romero Villagómez	Segunda versión
Plan del Proyecto v3.0.pdf	3.0	07/02/2015	Ing. Ricardo Romero Villagómez	Documento final

Contenido

1. Introducción

1.1 Propósito

El propósito del documento es dar a conocer de manera general los objetivos, recursos, alcances y limitaciones para el desarrollo del sistema MetricasCKMOOD.

1.2 Alcance

- Especificar los objetivos generales y específicos del sistema
- Especificar las funciones principales a implementar
- Determinar los recursos necesarios de hardware y software para la elaboración del sistema

1.3 Objetivos del Proyecto

1.3.1 Objetivo General

Desarrollar un componente para servidores de integración continua, que mediante herramientas de análisis estático, recolecte información de proyectos de software que apliquen el paradigma de la programación orientada a objetos en lenguaje Java, para generar las métricas CK y MOOD, estas métricas exploran la calidad del código de componentes de software que han sido diseñados con dicho paradigma.

1.3.2 Objetivos Específicos

- Diseñar y construir un sistema que integre las herramientas de análisis estático para recolectar la información, para generar métricas CK y MOOD, y realizar el análisis correspondiente de ellas
- El sistema debe de ejecutarse en un ambiente distribuido, para ser configurado y ejecutado desde el servidor de integración continua.
- Presentar en una página web las métricas obtenidas por cada proyecto de la aplicación. La información mostrada será además de las métricas normales (número de líneas de código, número de paquetes, número de clases, número de métodos, entre otras), las métricas de CK (WMC, DIT, NOC, CBO, RFC, LCOM, LCOM1, LCOM2, MPC, DAC, SIZE2) y MOOD (PF, CRF, MHF, AHF, MIF, AIF)
- Proporcionar diferentes formatos para exportar el análisis realizado, los formatos serán CSV, PDF y XLS

2. Organización del Proyecto

2.1 Integrantes

Nombre	Rol
Ing. Ricardo Romero Villagómez	Analista, Diseñador, Ingeniero y Arquitecto de Software
Dr. Carlos Alberto Fernández y Fernández	Asesor

2.2 Software

Herramienta	Uso
Enterprise Architect versión 10	Diseño y generación de diagramas en UML
Java (JDK) versión 1.7	Lenguaje de programación
Jenkins versión 1.633	Servidor de integración continua
Subversion versión 1.9	Herramienta de control de versiones
Maven versión 3	Herramienta de construcción automatizada
Eclipse Luna	Ambiente de desarrollo
VirtualBox versión 4.3.20	Herramienta de virtualización
CentOs versión 7	Sistema operativo para maquinas virtualizadas
MySQL versión 5.5	Base de datos
Tomcat versión 7	Servidor de Aplicaciones
JUnit versión 4.11	Pruebas unitarias

2.3 Hardware

Computadora Dell Optitplex 9020 CQ45 con las siguientes características:

- 8 MB en RAM
- Procesador Intel i7-4785 de 64 bits.
- Sistema Operativo: Windows 7

3. Prácticas y Medidas del Proyecto

3.1 Planeación de Fases

El desarrollo se llevará a cabo en base a fases con una o más iteraciones en cada una de ellas. La siguiente tabla muestra la distribución de tiempos y el número de iteraciones de cada fase (para las fases de Construcción y Transición es sólo una aproximación preliminar)

Fase	Núm. de iteraciones	Duración
Inicio	2	2 semanas
Elaboración	2	2 semanas
Construcción	2	5 y 3 semanas
Transición	2	1 Semana

3.2 Seguimiento y control del proyecto

3.2.1 Gestión de Requerimientos

Los requerimientos del sistema son especificados en los artefactos de la fase de inicio. Cada requisito tendrá una serie de atributos tales como importancia, estado, iteración donde se implementa, etc. Estos atributos permitirán realizar un efectivo seguimiento de cada requerimiento.

3.2.2 Control de Entregas

El calendario del proyecto tendrá un seguimiento y evaluación semanal por parte del asesor del proyecto.

3.2.3 Control de Calidad

Los defectos detectados en las revisiones tendrán un seguimiento para asegurar la conformidad respecto de la solución de dichas deficiencias Para la revisión de cada artefacto y su correspondiente garantía de calidad se utilizarán las guías de revisión y listas de verificación incluidas en la metodología *OpenUP*.

3.2.4 Gestión de Riesgos

A partir de la fase de Inicio se mantendrá una lista de riesgos asociados al proyecto y de las acciones establecidas como estrategia para mitigarlos o acciones de contingencia. Esta lista será evaluada al menos una vez en cada iteración.

3.2.5 Gestión de la Configuración

Se realizará una gestión de configuración para llevar un registro de los artefactos generados y sus versiones. También se incluirá la gestión de las modificaciones que se produzcan. Al final de cada iteración se establecerá una línea base (un registro del estado de cada artefacto, estableciendo una versión).

4. Hitos y Objetivos del Proyecto

4.1 Hitos

Fase	Hito
Inicio	En esta fase desarrollará los requisitos del producto desde la perspectiva de cubrir con los objetivos planteados en este documento, en el artefacto Visión del Proyecto y en el artefacto Requerimientos del Proyecto. Los principales casos de uso serán identificados y se hará en caso necesario, un refinamiento en los artefactos anteriores. La aceptación del asesor marcarán el final de esta fase
Elaboración	En esta fase se analizan los requerimientos tanto funcionales como los no funcionales y se desarrolla un prototipo de arquitectura (incluyendo las partes más relevantes y/o críticas del sistema). Al final de esta fase, todos los casos de uso correspondientes a los requerimientos que serán implementados en la primera versión de la fase de Construcción deben estar analizados y diseñados. La revisión y aceptación del prototipo de la arquitectura del sistema marca el final de esta fase. La primera iteración tendrá como objetivo la identificación y especificación de los principales casos de uso, así como su realización preliminar en el modelo de análisis/diseño, también permitirá hacer una revisión general del estado de los artefactos hasta este punto y ajustar si es necesario, los artefactos y/o la planificación del proyecto para asegurar el cumplimiento de los objetivos planteados.
Construcción	Durante la fase de construcción se terminan de analizar y diseñar todos los casos de uso, refinando en el modelo de análisis/diseño. El producto se construye en base a 2 iteraciones, cada una produciendo una versión a la cual se le aplican las pruebas y se valida el resultado generado. El hito que marca el fin de esta fase es la versión del código fuente del sistema, con la capacidad operacional parcial del producto que se haya considerado en los objetivos del proyecto.

Fase	Hito
Transición	En esta fase se preparará la versión para su distribución. El hito que marca el fin de esta fase incluye la entrega de toda la documentación del proyecto con los manuales de instalación, configuración y manual de usuario.

4.2 Objetivos

Iteración	Objetivos
I1	a) Análisis de librerías para la generación de métricas de la clasificación de código abierto existentes en el mercado b) Diseñar y construir el núcleo del sistema que recolecte la información del código fuente, para generar las métricas CK y MOOD c) Crear el *plugin* para el servidor de integración continua d) Presentar las métricas obtenidas en una página web
I2	a) Construir la funcionalidad necesaria para exportar las métricas obtenidas en los formatos antes especificados b) Realizar el refinamiento de la parte visual y de la usabilidad de la aplicación

5. Publicación de la Solución

El esquema de implantación del software propuesta para este proyecto es una fusión de 2 esquemas, catalogadas como implantación de bajo riesgo e implantación simple, para garantizar el éxito del proyecto, rapidez del proceso y utilización del menor volumen de recursos posibles, ya que pueden ser aplicadas en proyectos de alcance medio y de baja o media complejidad.

Entre las ventajas que se desean aportar al usar estos modelos, por mencionar algunas, son la rapidez en la implementación; estimación de bajo costo; implantación de bajo riesgo; altos beneficios; resultados predecibles; conocimiento muy alto de las características del sistema; integración; percepción del sistema como algo propio.

MetricasCKMOOD
Requerimientos del Proyecto

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Requerimientos del Proyecto.pdf	0.1	03/12/2015	Ing. Ricardo Romero Villagómez	Borrador
Requerimientos del Proyecto v1.0.pdf	1.0	07/12/2015	Ing. Ricardo Romero Villagómez	Primera versión
Requerimientos del Proyecto v2.0.pdf	2.0	15/12/2015	Ing. Ricardo Romero Villagómez	Segunda versión
Requerimientos del Proyecto v3.0.pdf	3.0	07/02/2015	Ing. Ricardo Romero Villagómez	Documento final

Contenido

1. Introducción

La medición de la calidad del software es una necesidad para las empresas de desarrollo de software ya que representa una ventaja estratégica al proporcionar el conocimiento de los procesos productivos y permitir mejoras en las tareas menos eficientes, ya que siempre habrá elementos cualitativos para la creación de software.

Las métricas de software tienen un papel decisivo en la obtención de un producto de alta calidad, porque determinan mediante estadísticas basadas en la experiencia, el avance del software y el cumplimiento de parámetros requeridos.

Este documento describe los requerimientos para el producto denominado MetricasCKMOOD, el cual pretende contribuir a la calidad de proyectos de software diseñados con el paradigma orientado a objetos, automatizando el proceso de verificación de métricas de código, con el objetivo de evaluar, de forma cuantitativa, si ciertas propiedades deseables del diseño orientado a objetos son cumplidas.

1.1 Descripción del Problema

En los últimos años hemos visto un incremento considerable del uso del paradigma orientado a objetos en el desarrollo de software. El uso de técnicas de desarrollo de software orientado a objetos introduce nuevos elementos a la complejidad del software, tanto en el proceso de desarrollo de software como en el producto final.

Muchas métricas orientadas a objetos se han propuesto específicamente con el fin de evaluar el diseño de un sistema de software. Sin embargo, la mayoría de los enfoques existentes para la medición de estos parámetros de diseño implican sólo algunos de los aspectos del paradigma orientado a objetos.

Las métricas aquí planteadas (CK y MOOD) tratan de captar los diferentes aspectos del software como producto y su proceso de desarrollo. Estas métricas se pueden aplicar en varios proyectos para evaluar y comparar el rendimiento del código utilizando el paradigma orientado a objetos.

1.2 Objetivo

Objetivo	Razón	Métrica
Contar con una implementación genérica para el análisis de métricas de diseño	Tener un código base para aplicarse en diversos proyectos de software	Reducir el % de esfuerzo para aplicar el análisis de métricas de diseño en diversos contextos.
Gestionar las métricas y el flujo entorno al análisis de métricas de diseño	Apoyar en la toma de decisiones sobre el diseño de software aplicando el paradigma orientado a objetos	Reducir el tiempo en la toma de decisiones
Realizar el análisis de forma constante	Obtener las métricas del código base de proyectos de software en lenguaje Java	Reducir el riesgo, ya que se pueden detectar desviaciones del diseño en etapas tempranas
Generar una base de conocimiento sólida y confiable	Explotar la información a través de herramientas de análisis	Tamaño de base de datos que permita hacer inferencias con alto grado de significado estadístico

2. Requerimientos Funcionales

2.1 RF-01 Obtener la métrica WMC

Nombre	RF-01 Obtener métrica WMC
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	4/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica WMC mide la complejidad de una clase. Clases con un gran número de métodos requieren más tiempo y esfuerzo para desarrollarlas y mantenerlas, ya que influirán en las subclases que heredan todos sus métodos. Definición: Considere una clase C_i con los métodos $M_1, M_2, \ldots, M_n$. Sean $c_1, c_2, \ldots, c_n$ la WMC de los métodos, entonces: $WMC = \sum_{i=1}^{n} c_i$

2.2 RF-02 Obtener la métrica DIT

Nombre	RF-02 Obtener la métrica DIT
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	4/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica DIT mide el máximo nivel en la jerarquía de herencia. Es el número de niveles en la jerarquía de herencia de una clase. Definición: La métrica DIT de una clase A es su profundidad en el árbol de herencia. Si A se encuentra en situación de herencia múltiple, la longitud máxima hasta la raíz será el DIT.

2.3 RF-03 Obtener la métrica NOC

Nombre	RF-03 Obtener la métrica NOC
Sistema	MetricasCKMOOD

Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	4/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica NOC es el número de subclases subordinadas a una clase en la jerarquía de herencia, es decir, el número de subclases que pertenecen a una clase. Es un indicador del nivel de reúso, la posibilidad de haber creado abstracciones erróneas y es un indicador del nivel de prueba requerido. Definición: El NOC de una clase es el número de subclases subordinadas a una clase en la jerarquía

2.4 RF-04 Obtener la métrica CBO

Nombre	RF-04 Obtener la métrica CBO
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	4/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica CBO de una clase es el número de clases con las que está acoplada. Una clase está acoplada a otra si utiliza sus métodos o variables de instancia, excluyendo el acoplamiento por herencia. Definición: El CBO de una clase, es el número de clases con la que está acoplada. Una clase está acoplada a otra, si utiliza sus métodos y variables de instancia, excluyendo el acoplamiento por herencia

2.5 RF-05 Obtener la métrica RFC

Nombre	RF-05 Obtener la métrica RFC
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	4/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase

Descripción de la Funcionalidad	La métrica RFC es el cardinal del conjunto de todos los métodos que pueden ser invocados en respuesta a un mensaje a un objeto de la clase o por algún método en la clase. Esto incluye todos los métodos accesibles dentro de la jerarquía de la clase. Definición: El *RFC* de una clase es el conjunto de métodos que se pueden ejecutar como respuesta a un mensaje recibido por un objeto de la clase. $RFC=\|RS\|$ RS es la RFC de la clase, dado que $RS=\{M\}\cup_{all\,i}\{R_i\}$, donde $\{R_i\}$ es el conjunto de métodos invocados por el método i y $\{M\}$ es el conjunto de todos los métodos de la clase.

2.6 RF-06 Obtener la métrica LCOM

Nombre	RF-06 Obtener la métrica LCOM
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica LCOM establece en qué medida los métodos hacen referencia a atributos. Es una medida de la cohesión de una clase teniendo en cuenta el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase. Definición: Considere una clase C_i con métodos $M_1, M_2, \ldots, M_n$. Sea $\{I_j\}$ el conjunto de variables de instancia usados por el método M_i. Existen n de esos conjuntos $\{I_1, I_2, \ldots, I_n\}$, sea $P=\{(I_i,I_j)\vee I_i\cap I_j=0\}$ y $Q=\{(I_i,I_j)\vee I_i\cap I_j\neq 0\}$ entonces: $LCOM=\|P\|-\|Q\|, si\,\|P\|>\|Q\|;\ de\ otra\ forma\ LCOM=0.$

2.7 RF-07 Obtener la métrica LCOM1

Nombre	RF-07 Obtener la métrica LCOM1
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase

Descripción de la Funcionalidad	La métrica LCOM1 establece otra medida en qué los métodos hacen referencia a atributos. Es una medida de la cohesión de una clase teniendo en cuenta el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase. Definición: LCOM1 = Número de conjuntos disjuntos de métodos locales en una clase. Cada conjunto tiene uno o más métodos locales de una clase, y cualquiera de los métodos en el acceso conjunto al menos un atributo de la clase común; el número de atributos comunes que van de 0 a N (donde N es un número entero mayor que 0).

2.8 RF-08 Obtener la métrica MPC

Nombre	RF-08 Obtener la métrica MPC
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	Esta métrica establece el número de métodos que son invocados en una clase. Se usa para medir la complejidad del paso de mensajes entre clases. Definición: MPC = Número de métodos invocados en una clase.

2.9 RF-09 Obtener la métrica DAC

Nombre	RF-09 Obtener la métrica DAC
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	Esta métrica permite obtener el número de atributos de una clase que tienen como tipo a otra clase. Como una clase puede verse como una implementación de tipos de datos abstractos, esto causa un tipo particular de acoplamiento. Definición: DAC = Número de atributos de una clase que tiene como tipo a otra clase.

2.10 RF-10 Obtener la métrica SIZE2

Nombre	RF-10 Obtener la métrica SIZE2

Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	Esta métrica define el número de propiedades (incluidos los atributos y métodos) definidos en una clase. Definición: SIZE2 = Número de atributos + número de métodos de una clase.

2.11 RF-11 Obtener la métrica LCOM2

Nombre	RF-11 Obtener la métrica LCOM2
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica LCOM2 es otra variante de la métrica LCOM, la cual definen como una normalización del número de métodos y variables de una clase. Definición: LCOM2 = (a – kl) / (l – kl). l = Número de atributos. k = Número de métodos. a = Sumatoria de los distintos atributo accedidos por cada método. Datos de Referencia: El resultado se da en un rango entre 0 y 1, donde el valor deseable de LCOM2 es 0.

2.12 RF-12 Obtener la métrica PF

Nombre	RF-12 Obtener la métrica PF
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase

Descripción de la Funcionalidad	La métrica PF representa el número real de posibles situaciones polimórficas diferentes, también representa el número máximo de posibles situaciones polimórficas diferentes para una clase en particular. Es además, una medida indirecta de la asociación dinámica del sistema. Definición: $$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) * DC(C_i)]}$$ Donde: $M_o(C_i) = M_n(C_i) + M_o(C_i)$ $M_n(C_i)$ = Número de métodos definidos en la clase C_i $M_o(C_i)$ = Número de métodos sobrecargados de la clase C_i $DC(C_i)$ = Número de descendientes (hijos) de la clase C_i TC = Total de clases

2.13 RF-13 Obtener la métrica CF

Nombre	RF-13 Obtener la métrica CF
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica CF se evalúa como una fracción, donde el numerador representa el número de acoplamientos no relacionados con la herencia y el denominador es el número máximo de acoplamientos en un sistema. El número máximo de acoplamientos incluye tanto la herencia y el acoplamiento no relacionada con la herencia. Los acoplamientos basados en herencia surgen como clases derivadas (subclases) y de la herencia de métodos y atributos forman su clase padre (superclase). Definición: $$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_{client}(C_I, C_J) \right]}{TC}$$ Donde: $is_{client}(C_i, C_j)$ = la relación entre la clase cliente y la clase objeto TC = total de clases

2.14 RF-14 Obtener la métrica MHF

Nombre	RF-14 Obtener la métrica MHF
Sistema	MetricasCKMOOD

Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	5/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica MHF mide la invisibilidad de los métodos en las clases. La invisibilidad de un método es el porcentaje de las clases totales de la que el método no es visible. Esta métrica es una fracción donde el numerador es la suma de todos los métodos ocultos definidos en el sistema. El denominador es el número total de los métodos definidos en el sistema. Definición: $MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$ Donde: $M_d(C_i) = M_v(C_i) + M_h(C_i)$ $M_d(C_i)$ = Número de métodos definidos en la clase C_i $M_v(C_i)$ = Número de métodos visibles de la clase C_i $M_h(C_i)$ = Número de métodos ocultos de la clase C_i TC = Total de clases

2.15 RF-15 Obtener la métrica AHF

Nombre	RF-15 Obtener la métrica AHF
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase

Descripción de la Funcionalidad	La métrica AHF mide la invisibilidad de los atributos de las clases. La invisibilidad de un atributo es el porcentaje de las clases totales de la que el atributo no es visible. Un atributo se llama visible si se puede acceder por otra clase u objeto. Los atributos deben ser ocultados en una clase. Se pueden conservar el acceso de otros objetos al ser declarada como privado. Es una fracción. El numerador es la suma de todos los atributos no visibles definidos en todas las clases. El denominador es el número total de atributos definidos en el sistema. Definición: $$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$ Donde: $A_d(C_i) = A_v(C_i) + A_h(C_i)$ $A_d(C_i)$ = Número de atributos definidos en la clase C_i $A_v(C_i)$ = Número de atributos visibles de la clase C_i $A_h(C_i)$ = Número de atributos ocultos de la clase C_i TC = Total de clases

2.16 RF-16 Obtener la métrica MIF

Nombre	RF-16 Obtener la métrica MIF
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica MIF es la suma de los métodos heredados en todas las clases del sistema. Mide el grado en que una clase en una arquitectura orientada a objetos hace uso de la herencia de métodos y atributos. Definición: $$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$ Donde: $M_a(C_i) = M_d(C_i) + M_i(C_i)$ $M_a(C_i)$ = Número de métodos definidos en la clase C_i $M_d(C_i)$ = Número de métodos declarados de la clase C_i $M_i(C_i)$ = Número de métodos heredados de la clase C_i TC = Total de clases

2.17 RF-17 Obtener la métrica AIF

Nombre	RF-17 Obtener la métrica AIF

Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Código Fuente
Salida:	Guardar la información obtenida por la métrica clase por clase
Descripción de la Funcionalidad	La métrica AIF es la relación de la suma de atributos heredados en todas las clases del sistema entre el número total de atributos disponibles para todas las clases. Expresa el nivel de reutilización en un sistema. Definición: $AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$ Donde: Aa(Ci) = Ad(Ci) + Ai (Ci) Aa(Ci) = Número de atributos definidos en la clase Ci Ad(Ci) = Número de atributos declarados de la clase Ci Ai (Ci) = Número de atributos heredados de la clase Ci TC = Total de clases

2.18 RF-18 Obtener Métricas CK

Nombre	RF-18 Obtener Métricas CK3
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Solicitud de ejecución
Salida:	Guardar la información de las métricas CK
Descripción de la Funcionalidad	Este requerimiento debe de lanzar el análisis de las métricas CK, las métrica a ejecutar son: WMC, DIT, NOC, CBO, RFC, LCOM, LCOM1, LCOM2, MPC, DAC y SIZE2

2.19 RF-19 Obtener Métricas MOOD

Nombre	RF-19 Obtener Métricas MOOD
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Solicitud de ejecución
Salida:	Guardar la información de las métricas MOOD
Descripción de la Funcionalidad	Este requerimiento debe de lanzar el análisis de las métricas MOOD, las métrica a ejecutar son: PF, CF, MHF, AHF, MIF y AIF

2.20 RF-20 Configurar Plugin

Nombre	RF-20 Configurar Plugin
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Solicitud de ejecución
Salida:	Guardar la información del plugin por default de las métricas CK y MOOD
Descripción de la Funcionalidad	Este requerimiento debe de cambiar a petición del usuario, la activación o desactivación del análisis de las métricas CK y MOOD. Una vez cambiados estos valores, la siguiente ejecución de la obtención de las métricas los debe de tomar como base para la presentación de la información generada por el análisis.

2.21 RF-21 Exportar Métricas

Nombre	RF-21 Exportar Métricas
Sistema	MetricasCKMOOD
Responsable:	Ing. Ricardo Romero Villagómez
Fecha:	6/12/2015
Entrada:	Solicitud de ejecución
Salida:	Resultado del análisis en diversos formatos
Descripción de la Funcionalidad	El resultado del análisis realizado se debe de exportar a tres formatos diferentes, él formato será escogido por el usuario. Los formatos en los que se podrá exportar la información son: CSV, PDF y XLS.

3. Atributos de Calidad

3.1 Usabilidad

No. de Requerimiento No Funcional	Descripción
RNF-01	La aplicación puede ser accedida a través de una interfaz diseñada en ambiente web.
RNF-02	Los mensajes de error deben ser reportados por la propia aplicación en la medida de las posibilidades y no por el Sistema Operativo. Estos mensajes de error deben ser informativos y orientados al usuario final.
RNF-03	El idioma de la aplicación debe ser multilenguaje, los idiomas principales serán español e inglés.
RNF-04	El acceso a las funcionalidades de la aplicación debe utilizar tipos de datos estándar de Internet.
RNF-05	La aplicación web debe poseer un diseño adaptativo (*Responsivo*) a fin de garantizar su adecuada visualización en múltiples computadores personales.

3.2 Confiabilidad

No. de Requerimiento No Funcional	Descripción
RNF-06	La aplicación debe ser tolerante ante los fallos y las operaciones a realizar deben ser transaccionales.
RNF-07	La aplicación debe tener una disponibilidad superior al 75%.
RNF-08	La probabilidad de falla de la aplicación no podrá ser mayor al 5%.
RNF-09	La aplicación debe ser capaz de operar adecuadamente con 1,000 usuarios concurrentes.

3.3 Compatibilidad

No. de Requerimiento No Funcional	Descripción
RNF-10	Se documentará la aplicación con un manual de ayuda con el objetivo de explicar el uso de la plataforma para garantizar el soporte de la herramienta.
RNF-11	Se debe realizar el proyecto de forma versionable que permita darle mantenimientos al sistema a fin de aumentar las funcionalidades y/o corregir los errores del mismo a través de versiones posteriores.
RNF-12	La aplicación debe poderse ejecutar en diferentes entornos, por lo menos para Windows 7, Linux 7 distribución centOS (Multiplataforma).
RNF-13	La aplicación se acogerá a las reglas de las licencias generales públicas de software libre (GNU GPL v3.0), es decir el usuario tiene la libertad de usarlo para cualquier propósito, tiene la libertad de cambiarlo para satisfacer sus necesidades, tiene la libertad de compartir el software y por último, tiene la libertad de compartir los cambios que realice al software.

4. Interfaces del sistema

4.1 Interfaces de usuario

La aplicación debe proveer interfaces que faciliten la interacción de los usuarios con la aplicación, estas interfaces deberán estar enfocadas a permitir la ejecución de los casos de uso definidos. Estas interfaces son:

- Configuración de valores de referencia de las métricas CK y MOOD
- Consulta de las métricas obtenidas

4.1.1 Look & Feel

- Dado que la aplicación será ejecutada desde un servidor de integración continua, se debe contemplar el diseño basado en páginas web que debe presentar el look and feel acorde al servidor de integración continua en la versión usada; de ser posible usar las hojas de estilo del servidor para reducir el tiempo de desarrollo de las hojas de estilo y evitar modificaciones futuras debido a la actualización del software de integración continua.
- Las páginas web deben de ser autoajustable a cualquier tamaño y resolución de pantalla del usuario, debe utilizar imágenes optimizadas y componentes de diseño que permitan mostrar la información de manera dinámica, ágil y estética.

4.1.2 Diseño y Requisitos de Navegación

- La aplicación debe de navegar en los exploradores más comunes como Mozilla, Internet Explorer, Chrome y las diferentes plataformas (Windows, Mac, Linux).
- El navegador no debe requerir ninguna modificación o instalación de *plugins*, *applets*, o similares para que el software funcione, ni requerir soporte técnico al usuario para poder operar la aplicación.
- La aplicación se debe de ajustar a las dimensiones que disponga el contenedor de página, del servidor de integración continua.

4.2 Interfaces con Sistemas Externos o Dispositivos

4.2.1 Interfaces de Software

- La aplicación necesita de la base de datos MySQL, la cual usará el JDBC como interfaz entre la base de datos y la aplicación.
- La aplicación usará un *plugin* con el servidor de integración continua Jenkins para que sea ésta la herramienta que ejecute las peticiones de cambio de valores de referencia y del análisis, generación y posteriormente el despliegue de las métricas obtenidas.

4.2.2 Interfaces de Hardware

En el caso de contar con una conexión a Internet, aplicación podrá funcionar en un ambiente conectado a partir de un modelo de sincronización entre sistemas. En este caso, las interfaces hardware con las que debe contar el usuario son:

- Tarjeta de red
- *WiFi* (opcional)
- Conexión banda ancha (preferentemente)

4.2.3 Interfaces de Comunicaciones

Las interfaces de comunicación deben contener los estándares Web y fundamentalmente se deben basar en el protocolo HTTP para la comunicación con los usuarios finales.

Para la comunicación con las diferentes herramientas es necesario el uso del protocolo TCP/IP.

5. Reglas de Negocio

No aplica

6. Limitaciones del Sistema

- El análisis de las métricas CK y MOOD se realizará en proyectos desarrollados en lenguaje Java.
- La ejecución de la aplicación se realizará única y exclusivamente por medio del servidor de integración continua.

7. Documentación del Sistema

La creación de la documentación de la aplicación queda en la responsabilidad del ingeniero de software, así como su definición y administración. La documentación debe ser clasificada en documentación técnica y documentación funcional, los cuales estarán integrados por los siguientes manuales:

- Documentación técnica (manual técnico y de instalación)
- Documentación funcional (manual de configuración y de usuario final

APÉNDICE 2. ENTREGABLES FASE ELABORACIÓN

Los documentos que componen esta sección son:

Arquitectura del proyecto

MetricasCKMOOD
Documento de Arquitectura

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Arquitectura v1.0.odt	1.0	17/12/2015	Ing. Ricardo Romero Villagómez	Borrador
Arquitectura v2.0.odt	2.0	01/03/2016	Ing. Ricardo Romero Villagómez	Actualización
Arquitectura v3.0.odt	3.0	17/03/2016	Ing. Ricardo Romero Villagómez	Actualización
Arquitectura v4.0.odt	4.0	07/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1. Propósito

El propósito de este documento es consolidar toda la información relevante al diseño de alto nivel para el sistema MetricasCKMOOD. El documento contiene diversos artefactos que acompañan la arquitectura, incluyendo el diagrama de contexto, el diseño de pantallas y un diagrama entidad-relación.

Para la fase de desarrollo, se tiene contemplado el desarrollo de la configuración de las métricas en un servidor de integración continua y la presentación de las métricas obtenidas, los cuales consistirán en la construcción de pantallas que podrán ser visualizadas en una página web por medio de un navegador, desde la página principal del servidor de integración continua, la exportación de las métricas obtenidas en forma de reportes generados en formato CSV, PDF y XLS.

Al ser construido como un *plugin* de un servidor de integración continua (Jenkins), el sistema se debe de adaptar a los puntos de extensibilidad definidos por Jenkins, estos puntos de extensibilidad son interfaces o clases abstractas que modelan un aspecto de un sistema de construcción. Esas interfaces de definen contratos de lo que se necesitan ser implementado.

Las limitaciones para construir el *plugin* definidas por Jenkins son el uso de la herramienta *Maven* 3 y JDK 6.0 o posterior.

2. Objetivos de la arquitectura

a) Definir los módulos principales

b) Definir las responsabilidades de cada módulo

c) Definir la interacción entre cada módulo

d) Definir los protocolos de comunicación e interacción

e) Planificar la evolución del sistema, identificando las partes mutables e inmutables

f) Analizar el grado de cumplimiento de los requerimientos iniciales

3. Supuestos y dependencias

a) Se creará el plugin de acuerdo a las especificaciones definidos por *SonarQube*.

b) El sistema a desarrollar se deberá de ejecutar en una herramienta colaborativa llamada *Jenkins*.

4. Requerimientos arquitectónicos significativos

Escenario	Importancia para negocio	Dificultad de implementación	Prioritario
Garantizar la disponibilidad de la información	Alta	Alta	Si
Garantizar la integridad de la información	Alta	Alta	Si
Garantizar la consistencia de la información	Alta	Media	Si
Soportar 1000 usuarios de forma concurrente	Alta	Alta	Si
Soporte para bases de datos *MySQL*	Alta	Media	Si
El sistema debe ser fácil de usar.	Alta	Alta	Si

5. Decisiones, limitaciones y justificaciones

Restricciones de negocio

5.1 Integración con Sistema de Control de Versiones

Declaración	El sistema deberá permitir la integración con el Sistema de Control de Versiones configurado para obtener la información de versionamiento del activo de software a analizar.
Justificación	Para conocer integralmente la evolución del activo durante su vida útil.
Implicaciones	Esta información debe estar sincronizada con el sistema de control de versiones y se debe poder tener información como: versiones liberadas, últimos cambios, quienes realizaron los cambios, estado.

5.2 Manejo de alto volúmenes de información

Declaración	El sistema debe permitir el manejo de altos volúmenes de información.
Justificación	Manejo de la información de todos los activos de software de un proyecto en lenguaje Java.
Implicaciones	Diseñar un modelo de datos que soporte los altos volúmenes de transacciones. Diseñar las integraciones con los sistemas externos de manera que no afecten la eficiencia de la aplicación. Definir políticas de gestión de información para evitar replicas innecesarias de la información.

5.3 Exportar información de reportes

Declaración	El sistema deberá permitir exportar la información de los análisis generados.
Justificación	Para facilitar una mejor comprensión de la información y apoyar la toma de decisiones se requiere que la información se disponga de manera legible para los interesados.
Implicaciones	Desarrollar funcionalidades que permitan a los interesados exportar la información almacenada en la base de datos a los formatos CSV, PDF y XLS.

Restricciones técnicas

5.4 Plataforma colaborativa

Declaración	El sistema deberá estar orientado a ser una plataforma colaborativa.
Justificación	Proveer un mecanismo para mantener en forma integrada la información originada en proyectos creados en lenguaje Java. Permitir a todos los usuarios de la herramienta de integración continua, la ejecución de tareas con el fin de actualizar la información de los análisis realizados.
Implicaciones	Integrar a la herramienta de integración continua, un componente de software que facilite el logro de esta restricción.

5.5 Lenguaje de implementación

Declaración	El sistema debe ser web, utilizando tecnologías de código abierto, y el lenguaje de programación es Java.
Justificación	Disminuir el riesgo de dependencia del fabricante, facilitar el licenciamiento, bajo costo, alta seguridad.

Implicaciones	El uso de este tipo de tecnologías por lo general no cuenta con soporte oficial, por lo cual depende mucho del nivel de conocimiento de los involucrados en el proyecto.

5.6 Motor de base de datos

Declaración	La base de datos central del sistema deberá ser implementada en *MySQL*.
Justificación	Disminuir el riesgo de dependencia del fabricante, facilitar el licenciamiento, bajo costo, alta seguridad.
Implicaciones	Usar un librería flexible en la capa del modelo que permita usar el motor fácilmente, adicionalmente las consultas realizadas tendrían que estar hechas de forma simple (sin nada específico de cada motor).

6. Mecanismos arquitectónicos

6.1 Implementación 3-Capas

Declaración	Los componentes físicos de la aplicación se encontraran distribuidos en diferentes nodos "Unidades independientes de procesamiento", separando: las funcionalidades de negocio, los datos y la vista de la aplicación.
Justificación	Se desea sacar el mayor provecho, y obtener el mejor rendimiento de cada uno de los módulos que integraran la aplicación, además de los múltiples beneficios que esto ofrece.
Implicaciones	Independencia de ejecución, fácil escalabilidad, mejor procesamiento y aprovechamiento de los recursos de hardware, facilidad de mantenimiento. Reutilización de la funcionalidad por otros componentes o sistemas.

6.2 Estructura orientada a objetos

Declaración	Todos los componentes de la aplicación serán desarrollados haciendo uso del paradigma de desarrollo orientado a objetos.
Justificación	Basados en los objetos de negocio ya definidos se asignaran responsabilidades y la información específica propia del componente de acuerdo a los objetos que los conforman de manera coherente.
Implicaciones	Desacoplamiento entre los diferentes objetos del sistema, facilidad de mantenimiento, facilidad de escalar, facilidad para hacer pruebas y asegurar su correcto funcionamiento. El análisis global de la aplicación se realiza más fácil cuando es visto de manera independiente desde el funcionamiento y estructura de cada uno de sus elementos y luego asimilar como trabajan de manera integrada.

6.3 Estructura basada en componentes

Declaración	La funcionalidad de los diferentes módulos que conformaran la aplicación estará desarrollada por componentes.

Justificación	Se concentraran las funcionalidades relevantes a los objetos que los componen, se permitirá la comunicación entre objetos mediante interfaces expuestas de los diferentes componentes.
Implicaciones	Alta cohesión de los componentes que estructuran la aplicación, reutilización de las diferentes funcionalidades ofrecidas por los componentes, desacoplamiento entre los diferentes componentes del sistema, facilidad de mantenimiento, facilidad de escalar, facilidad para hacer pruebas y asegurar su correcto funcionamiento de los componentes de manera individual.

7. Vistas arquitectónicas

La arquitectura del sistema denota el conjunto de estructuras de alto nivel de un sistema de software. Estas estructuras están conformadas por elementos de software, las relaciones entre estos y las propiedades tanto de dichos elementos como de sus relaciones.

Dichas estructuras usualmente permiten visualizar un sistema desde diferentes perspectivas: lógica, operacional, de casos de uso. En este sentido, en las siguientes secciones se presentarán las estructuras de arquitectura que permitirán cubrir dichas perspectivas. Cada perspectiva tiene una notación particular de representación, que en nuestro caso usaremos el Lenguaje de Modelado Unificado (*UML*) para describir cada una.

7.1 Vista Lógica

Representa las capas del sistema distribuido, los componentes de servicio del sistema son descritos en niveles de servicio de infraestructura para proporcionar asistencia a los componentes de aplicaciones de todas las capas lógicas que se muestran en la Ilustración 1.

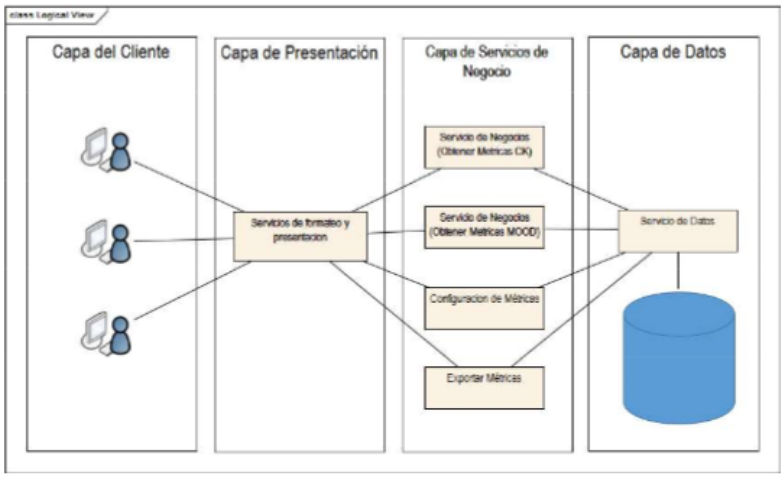


Ilustración 1: Vista Lógica

- Capa de cliente. La capa de cliente está formada por la lógica de la aplicación a la que el usuario final accede directamente mediante una interfaz de usuario. La lógica de la capa de cliente podría incluir clientes basados en navegadores web que sean ejecutados en un equipo de escritorio o portátil.
- Capa de presentación. La capa de presentación está formada por la lógica de aplicación, que prepara datos para su envío a la capa de cliente y procesa solicitudes desde la capa de cliente para su envío a la lógica de negocios del servidor, por medio de los servicios de formateo y presentación.
- Capa de servicios de negocios. La capa de servicios de negocio consiste en la lógica que realiza las 4 funciones principales de la aplicación: obtener métricas CK, obtener métricas MOOD, configuración de métricas y exportación de las métricas obtenidas. El servicio de negocio "obtener métricas CK" está dedicado a analizar el código para obtener las métricas WMC (Weighted Methods Per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling Between Object Classes), RFC (Response for Class), LCOM (Lack of Cohesion of Methods), MPC (Message Passing Coupling), DAC (Data Abstraction Coupling) y SIZE2; el servicio "obtener métricas MOOD" se avoca a analizar el código para obtener las métricas PF (Polymorphism Factor), CF (Coupling Factor), MHF (Method Hiding Factor), AHF (Attribute Hiding Factor), MIF (Method Inheritance Factor) y AIF (Attribute Inheritance Factor); el servicio "configuración de métricas" está enfocado a la validación y actualización de los valores por defecto de las métricas y por último, el servicio "exportación de las métricas" se encuentra dedicado a la recuperación de las métricas obtenidas y a la exportación de dichas métricas en los formatos ya establecidos.
- Capa de datos. La capa de datos está formada por los servicios que proporcionan los datos persistentes utilizados por la lógica de negocios y que son almacenados en un sistema de administración de bases de datos. Básicamente ésta capa realiza el almacenamiento de las métricas obtenidas en los proyectos java analizados.

La vista lógica (Ilustración 1) destaca la independencia lógica y física de los componentes, representada mediante 4 capas separadas. Estas capas representan la partición de la lógica de la aplicación en varios equipos en un entorno de red:

Independencia lógica. Las cuatro capas del modelo arquitectónico representan independencia lógica: puede modificar la lógica de la aplicación en una capa (por ejemplo, en la capa de servicio de negocios) independientemente de la lógica de las otras capas. Puede cambiar la implementación de la lógica de negocios sin tener que cambiar o actualizar la lógica de la capa de presentación o la de cliente. Esta independencia significa, por ejemplo, que puede introducir nuevos tipos de componentes de clientes sin tener que modificar los componentes de los servicios de negocios.

Independencia física. Las cuatro capas también representan independencia física: es posible implementar la lógica en capas distintas en varias plataformas de hardware (es decir, varias configuraciones de procesador, conjuntos de chips y sistemas operativos). Esta independencia permite ejecutar componentes de aplicación distribuida en los equipos que mejor se adapten a las necesidades informáticas individuales y a maximizar el ancho de banda de red.

7.2 Vista Operacional.

En la vista operacional (Ilustración 2) generalmente se muestran los nodos físicos del sistema, las capas, las tecnologías asociadas y los componentes que se ejecutan en cada nodo. Los elementos que aparecen en este diagrama en general permiten satisfacer los requerimientos no-funcionales (atributos de calidad) y se deben apegar a las posibles restricciones tecnológicas.

En el diagrama operacional se muestra:

- El nodo "Computadora Personal" es la que contiene el navegador web en la computadora del cliente y es donde se muestra información de la aplicación al cliente. Se comunica con el nodo "Servidor de Web" mediante el protocolo *HTTP* (*Hypertext Transfer Protocol*).
- El nodo "Servidor Web", es donde reside la parte estática de la aplicación. Se comunica con el nodo "Servidor de Aplicaciones" por medio del protocolo *HTTP*.
- El nodo "Servidor de Aplicaciones", es donde se publica y se configura la herramienta de IC (Jenkins), además es el nodo en donde se ejecutan las tareas para realizar el análisis de calidad del código. Se comunica con el nodo "Servidor Métricas CKMOOD" por medio del protocolo *HTTP*.
- El nodo "Servidor Métricas CKMOOD", es el que contendrá la aplicación a desarrollar y es la encargada de recolectarla información para la generación de las métricas CK y MOOD. Internamente contiene la capa de presentación, la capa de negocio y la capa de datos; estas tres capas se comunican mediante interfaces y únicamente la capa de datos es la que se comunica con el nodo "Servidor de Bases de Datos" por medio de la interface *JDBC* (*Java Database Connectivity*).
- Por último, tenemos el nodo "Servidor de Bases de Datos", este nodo contiene la herramienta de base de datos utilizada (MySQL) y es donde se almacena el resultado tanto del análisis realizado, como de las métricas obtenidas.

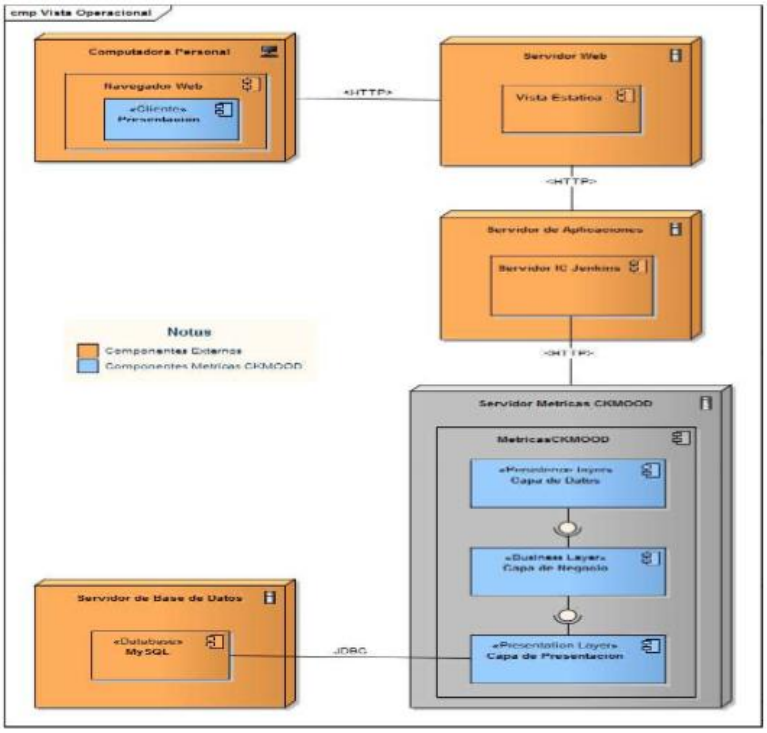


Ilustración 2: Vista Operacional

7.3 Diagrama de Casos de Uso.

El diagrama de casos de uso (Ilustración 3) contiene las interacciones del usuario con el sistema, además de especificar la interacción, comunicación y relación entre los elementos del modelo que cubren los requerimientos del sistema. En dicho diagrama se presentan 2 actores que interactúan con el sistema: usuario final y un actor interno llamado Jenkins.

Como se muestra en el diagrama de casos de uso, el actor "usuario final", va a poder modificar los valores por default de las métricas CK y MOOD por medio del caso de uso "Configurar Métricas"; y por medio del caso de uso "Exportar Métricas" va a poder descargar el resultado de las métricas en los formatos ya establecidos.

Para los casos de uso "Obtener Métricas", "Obtener Métricas MOOD" y "Obtener Métricas CK "el actor que lanza dichos procesos es un actor interno llamado "Jenkins", ya que dichos casos de uso serán ejecutados como resultado de una tarea de análisis dentro de la herramienta de IC.

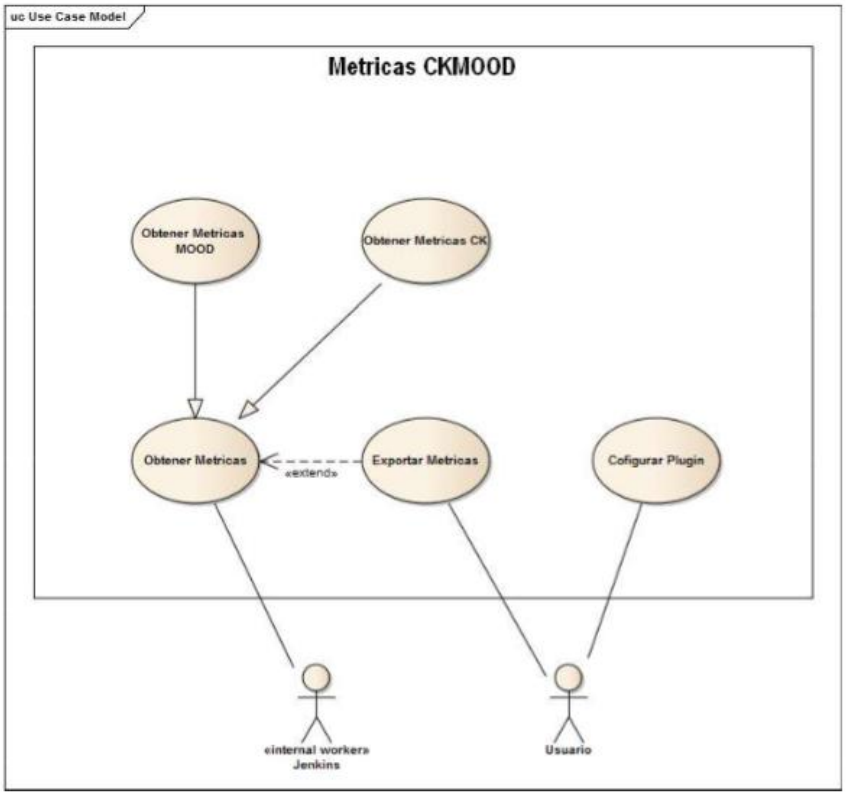


Ilustración 3 Diagrama de Casos de Uso

APÉNDICE 3. ENTREGABLES FASE CONSTRUCCIÓN

Los documentos que componen esta fase son:

Plan de iteración 1

Plan de iteración 2

Diseño detallado

Casos de uso

Caso de Uso: CU1 - Obtener métricas

Caso de Uso: CU2 – Obtener métricas CK

Caso de Uso: CU3 – Obtener métricas MOOD

Caso de Uso: CU4 – Exportar métricas

Caso de Uso: CU5 – Configurar métricas

Plan de pruebas

Casos de prueba

Código fuente

Plan de iteración 1

MetricasCKMOOD
Plan por iteración 1

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Plan por iteración1 v1.0.pdf	1.0	09/03/2016	Ing. Ricardo Romero Villagómez	Borrador
Plan por iteración1 v2.0.pdf	2.0	17/03/2016	Ing. Ricardo Romero Villagómez	Actualización
Plan por iteración1 v3.0.pdf	3.0	03/06/2016	Ing. Ricardo Romero Villagómez	Actualización
Plan por iteración1 v4.0.pdf	4.0	10/08/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

MetricasCKMOOD
Plan por iteración 1

1. Puntos Clave

Hito	Fecha
Inicio de Iteración 1	**07/03/2016**
Investigación de librerías	
Diseño y construcción del núcleo del sistema	
Construcción del plugin	
Presentación de las métricas obtenidas en pantalla	
Fin de Iteración 1	**11/04/2016**

2. Objetivos de Alto nivel

- Análisis de librerías para la generación de métricas de la clasificación de código abierto existentes en el mercado
- Diseñar y construir el núcleo del sistema que recolecte la información del código fuente, para generar las métricas CK y MOOD
- Crear el plugin para el servidor de integración continua
- Presentar el resultado de las métricas obtenidas en pantalla

3. Asignación de trabajo

Nombre o palabras clave de descripción	Prioridad	Estimación de tamaño *	Estatus	Iteración	Asignado a	Horas estimadas
Especificación de caso de uso CU1 - Obtener métricas	Alta	Simple	Asignado	1	RRV	2
Diseño detallado de CU1	Alta	Simple	Asignado	1	RRV	2
Casos de prueba de CU1	Alta	Simple	Asignado	1	RRV	2
Implementación de CU1	Alta	Simple	Asignado	1	RRV	10
Ejecución de casos de prueba de CU1	Alta	Simple	Asignado	1	RRV	2
Especificación de caso de uso CU2 – Obtener métricas CK	Alta	Complejo	Asignado	1	RRV	4
Diseño detallado de CU2	Alta	Complejo	Asignado	1	RRV	4
Casos de prueba de CU2	Alta	Complejo	Asignado	1	RRV	4
Implementación de CU2	Alta	Complejo	Asignado	1	RRV	20
Ejecución de casos de prueba de CU2	Alta	Complejo	Asignado	1	RRV	4
Especificación de caso de uso CU3 – Obtener métricas MOOD	Alta	Complejo	Asignado	1	RRV	4
Diseño detallado de CU3	Alta	Complejo	Asignado	1	RRV	4
Casos de prueba de CU3	Alta	Complejo	Asignado	1	RRV	4
Implementación de CU3	Alta	Complejo	Asignado	1	RRV	20
Ejecución de casos de prueba de CU3	Alta	Complejo	Asignado	1	RRV	4

* La estimación del tamaño se realiza por medio de la estimación Puntos Casos de Uso (*Use Case Points*), donde:

Categoría de caso de uso	Descripción	Peso
Simple	Interfaz de usuario sencilla que interactúa con una sola entidad de base de datos. Escenario con tres pasos o menos y su implementación implica menos de cinco clases.	5
Promedio	Interfaz de usuario con más diseño que interactúa con dos o más entidades de base de datos. Escenario desde cuatro hasta siete pasos y su implementación implica entre cinco y 10 clases.	10
Complejo	Interfaz de usuario muy compleja o es un procesamiento de datos, que interactúa con tres o más entidades de base de datos. Escenario con más de siete pasos y cuya implementación involucra a más de 10 clases.	15

4. Problemas detectados por iteración

Problema	Estatus	Notas
Falta de documentación para la generación de *plugins* para *SonarQube*	Resuelto	Debido a la falta de documentación formal sobre el desarrollo sobre la herramienta *SonarQube*, se recurrió a analizar el código de diversos *plugins* realizados por terceras personas. Esto ocasionó k se ampliara considerablemente el plazo para completar la iteración

5. Criterios de evaluación

- Construcción del 100% de la funcionalidad
- Más del 90% de casos de pruebas exitosos
- Obtención de las métricas CK y MOOD de forma correcta
- Presentación de las métricas obtenidas en una página web
- Integración correcta con el servidor de IC

6. Evaluación

Objetivo a evaluar	Valor obtenido de la métrica WMC
Fecha	06/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica DIT
Fecha	06/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica NOC
Fecha	06/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica CBO
Fecha	07/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica RFC
Fecha	07/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica LCOM
Fecha	07/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica LCOM1
Fecha	08/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica MPC
Fecha	08/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica DAC
Fecha	08/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica SIZE2
Fecha	08/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica LCOM2
Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica PF

MetricasCKMOOD
Plan por iteración 1

Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica CF
Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica MHF
Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica AHF
Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica MIF
Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Valor obtenido de la métrica AIF
Fecha	09/04/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

6.1 Evaluación con respecto a los objetivos

- Se realizó el análisis de librerías para la generación de métricas de la clasificación de código abierto existentes en el mercado, llegando a la conclusión de que no cubría con todas las métricas necesarias o que obtenían las métricas por otro método no definido en la especificación de esta investigación
- Se diseñó y se implementó el núcleo del sistema para recolectar la información del código fuente, y así generar las métricas CK y MOOD
- Se construyó el *plugin* para el servidor de integración continua, integrando el núcleo del sistema
- No se presentó el resultado de las métricas obtenidas en pantalla, en ésta iteración se comprobó únicamente que los valores de las métricas fueran almacenadas en la base de datos.

6.2 Planificación de la realidad en comparación con el trabajo terminado

- La planificación de las tareas fue errónea en relación a los tiempos estimados, se debió de dar más tiempo para la investigación de la construcción de *plugins* para *SonarQube*.
- No se tomó en cuenta la poca o nula información formal existente sobre el tema y como se recurrió a analizar el código de diversos *plugins* realizados por terceras personas, el tiempo dedicado a esta actividad no fue considerado. Esto ocasionó k se ampliara considerablemente el plazo para completar la iteración

6.3 Evaluación de los resultados de las pruebas

- Se llevaron a cabo 20 pruebas unitarias automatizadas, de las cuales en la primera iteración de pruebas se tuvo un éxito del 90%, mismas k después de realizar las modificaciones pertinentes se completó con éxito el 100% de las pruebas.

6.4 Preocupaciones y desviaciones

- N/A.

Plan de iteración 2

MetricasCKMOOD
Plan por iteración 2

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Plan por iteración v1.0.pdf	1.0	09/03/2016	Ing. Ricardo Romero Villagómez	Borrador
Plan por iteración v2.0.pdf	2.0	17/03/2016	Ing. Ricardo Romero Villagómez	Actualización
Plan por iteración v3.0.pdf	3.0	07/04/2016	Ing. Ricardo Romero Villagómez	Actualización
Plan por iteración v4.0.pdf	4.0	21/09/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

MetricasCKMOOD
Plan por iteración 2

1. Puntos Clave

Hito	Fecha
Inicio de Iteración 2	**12/04/2016**
Exportar las métricas obtenidas a los formatos especificados	
Crear funcionalidad para que el usuario configure los valores de referencia de las métricas	
Refinar la parte visual del sistema	
Fin de Iteración 2	**03/05/2016**

2. Objetivos de Alto nivel

- Construir la funcionalidad necesaria para exportar las métricas obtenidas en los formatos antes especificados
- Construir la funcionalidad para que el usuario configure el *plugin*
- Realizar el refinamiento de la parte visual y de la usabilidad de la aplicación

3. Asignación de trabajo

Nombre o palabras clave de descripción	Prioridad	Estimación de tamaño *	Estatus	Iteración	Asignado a	Horas estimadas
Especificación de caso de uso CU4 – Exportar métricas	Media	Promedio	Asignado	2	RRV	2
Diseño detallado de CU4	Media	Promedio	Asignado	2	RRV	2
Casos de prueba de CU4	Media	Promedio	Asignado	2	RRV	2
Implementación de CU4	Media	Promedio	Asignado	2	RRV	30
Ejecución de casos de prueba de CU4	Media	Promedio	Asignado	2	RRV	2
Especificación de caso de uso CU5 – Configurar métricas	Media	Promedio	Asignado	2	RRV	2
Diseño detallado de CU5	Media	Promedio	Asignado	2	RRV	2
Casos de prueba de CU5	Media	Promedio	Asignado	2	RRV	2
Implementación de CU5	Media	Promedio	Asignado	2	RRV	10
Ejecución de casos de prueba de CU5	Media	Promedio	Asignado	2	RRV	2

* La estimación del tamaño se realiza por medio de la estimación Puntos Casos de Uso (*Use Case Points*), donde:

Categoría de caso de uso	Descripción	Peso
Simple	Interfaz de usuario sencilla que interactúa con una sola entidad de base de datos. Escenario con tres pasos o menos y su implementación implica menos de cinco clases.	5
Promedio	Interfaz de usuario con más diseño que interactúa con dos o más entidades de base de datos. Escenario desde cuatro hasta siete pasos y su implementación implica entre cinco y 10 clases.	10
Complejo	Interfaz de usuario muy compleja o es un procesamiento de datos, que interactúa con tres o más entidades de base de datos. Escenario con más de siete pasos y cuya implementación involucra a más de 10 clases.	15

4. Problemas detectados por iteración

Problema	Estatus	Notas

5. Criterios de evaluación

- Construcción del 100% de la funcionalidad
- Más del 90% de casos de pruebas exitosos
- Exportación de las métricas obtenidas en los formatos CSV, PDF y XLS

6. Evaluación

Objetivo a evaluar	Generación del reporte en formato CSV
Fecha	03/05/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Generación del reporte en formato PDF
Fecha	03/05/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Generación del reporte en formato XLS
Fecha	03/05/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Validar la configuración del plugin
Fecha	03/05/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

Objetivo a evaluar	Usabilidad del componente
Fecha	03/05/2016
Participantes	Ricardo Romero Villagómez
Estatus del proyecto	Con atraso

6.1 Evaluación con respecto a los objetivos

- Se cubrió la construcción de la funcionalidad necesaria para exportar las métricas en los formatos CSV, PDF y XLS.
- Se implementó la funcionalidad para que el usuario pueda configurar si el *plugin* genere las métricas o no.

- Realizar el refinamiento de la parte visual y de la usabilidad de la aplicación

6.2 Planificación de la realidad en comparación con el trabajo terminado

- La planificación de las tareas fue exitosa, para ambos casos de uso. Aunque el proyecto tiene atraso, las tareas planeadas se realizaron en los tiempos esperados

6.3 Evaluación de los resultados de las pruebas

- Se llevaron a cabo 3 pruebas unitarias automatizadas, de las cuales en la primera iteración de pruebas se tuvo un éxito 100%.

6.4 Preocupaciones y desviaciones

- N/A.

MetricasCKMOOD
Diseño Detallado

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Diseño Detallado v1.0.pdf	1.0	25/03/2016	Ing. Ricardo Romero Villagómez	Borrador
Diseño Detallado v2.0.pdf	2.0	01/04/2016	Ing. Ricardo Romero Villagómez	Actualización
Diseño Detallado v3.0.pdf	3.0	22/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

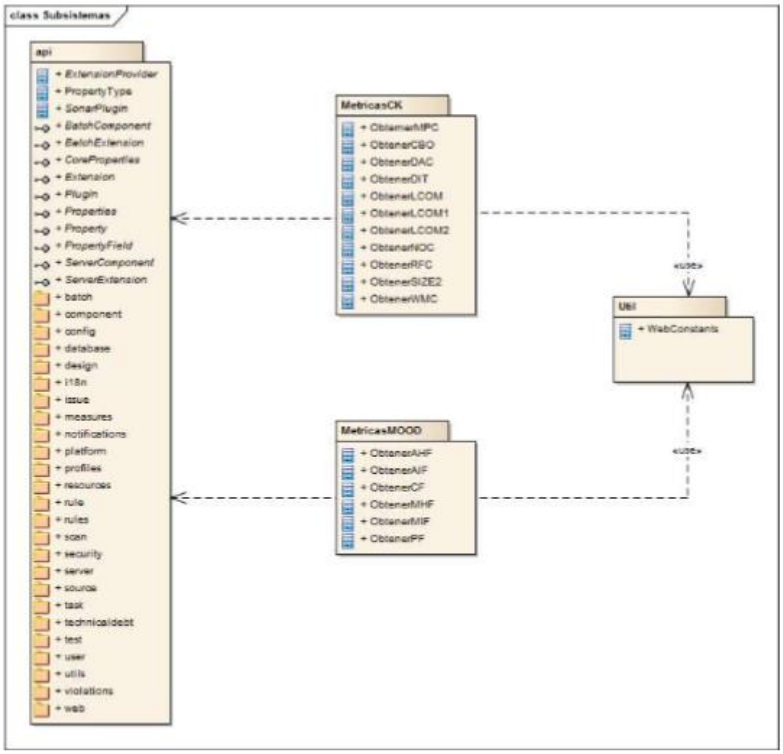
Contenido

MetricasCKMOOD
Diseño Detallado

1. Propósito

El propósito de este documento es presentar los subsistemas y patrones usados para la realización de la obtención de las métricas CK y MOOD de un proyecto java.

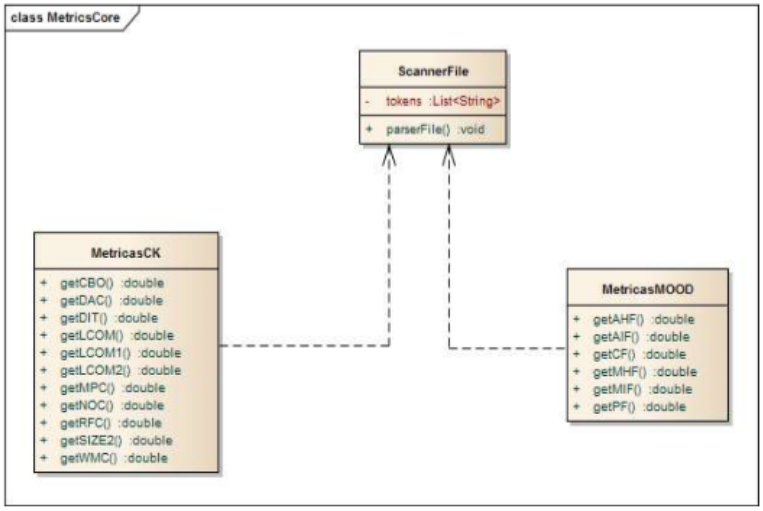
2. Estructura de Diseño



3. Subsistemas

Métricas Core

Este subsistema se encarga de la generación de las métricas CK y MOOD a cada componente del proyecto java. En este subsistema se encuentra la lógica de negocio de la generación de las métricas.



4. Patrones

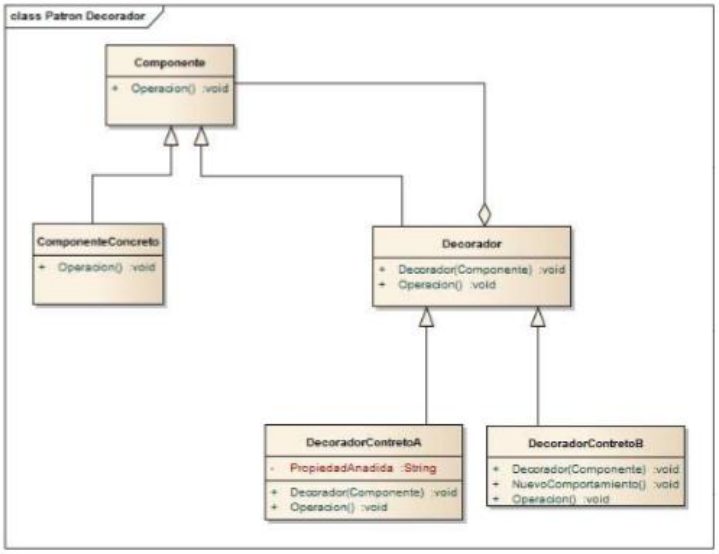
Decorador

4.1 Información general

La utilidad principal del patrón Decorador, es la de agregar funcionalidad de forma dinámica a objetos para ofrecer más funcionalidad de la que se tenía al principio. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera, esto es muy útil para evitar el uso de jerarquías de clases muy complejas.

Se aplica para añadir responsabilidad a objetos individuales de forma dinámica y transparente; cuando la extensión por medio de la herencia no es viable; cuando hay necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo por medio de herencia, cuando existe la necesidad de extender la funcionalidad de forma dinámica y posteriormente eliminar dicha funcionalidad.

4.2 Estructura



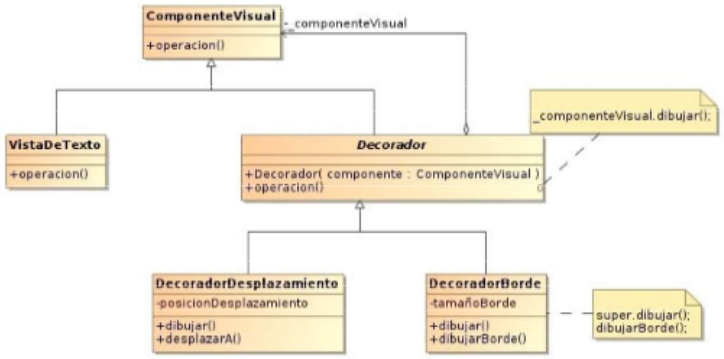
Participantes:

- **Componente**: Define la interfaz para los objetos que pueden tener responsabilidades añadidas.
- **ComponenteConcreto**: Define un objeto al cual se le pueden agregar responsabilidades adicionales.
- **Decorador**: Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.
- **DecoradorConcreto**: Añade responsabilidades al componente.

4.3 Comportamiento

Para mostrar el comportamiento de este patrón, se usará el siguiente ejemplo: Se tiene una herramienta para crear interfaces gráficas, que permite añadir funcionalidades como bordes o barras de desplazamiento a cualquier componente de la interfaz. La solución está en encapsular dentro de otro objeto, llamado Decorador, las nuevas responsabilidades. El decorador redirige las peticiones al componente y además, puede realizar acciones adicionales antes y después de la redirección. De este modo, se pueden añadir decoradores con cualidades añadidas recursivamente.

El diagrama de clases de la solución propuesta es el siguiente:

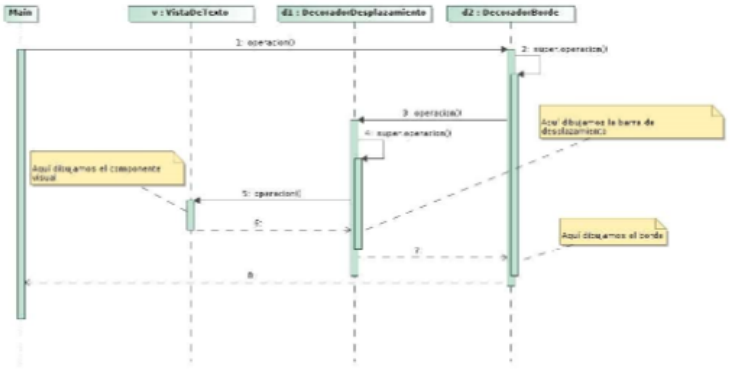


En este diagrama de clases, podemos ver que la interfaz decorador implementa la interfaz del componente, redirigiendo todos los métodos al componente visual que encapsula. Las subclases decoradoras refinan los métodos del componente, añadiendo o modificando responsabilidades.

Puntos relevantes a tomar en cuenta al usar este patrón:

- La interfaz de un objeto decorador debe ajustarse a la interfaz del componente que decora (por herencia).
- No hay necesidad de definir una clase Decorador abstracta cuando sólo tiene que añadir una responsabilidad.
- Es importante mantener la clase componente lo más liviana posible para evitar que los decoradores resulten demasiado cargados, es decir, debe centrarse en la definición de una interfaz, no en el almacenamiento de datos (quienes deberían ser tratados en subclases). Además, esta interfaz debe ser limitada, ya que si es amplia aumenta la probabilidad de que las subclases concretas terminen heredando funciones que no necesitan.
- Decorador vs Estrategia: son dos formas alternativas de cambiar un objeto. El patrón decorador sólo cambia un componente desde el exterior, de modo que los decoradores son transparentes para el componente. Con estrategia, el componente conoce las posibles extensiones, teniendo que hacer referencia y mantener las correspondientes estrategias.

El diagrama de secuencia es:



Se puede ver que el cliente no necesita hacer distinción entre los componentes visuales decorados y los no decorados.

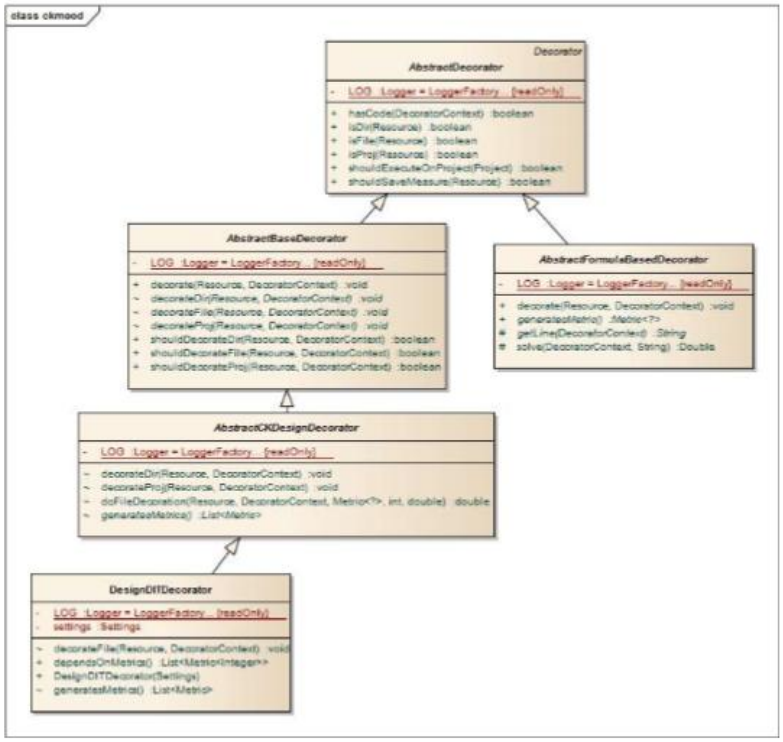
5. Realizaciones

Decorador DIT

5.1 Vista de los participantes

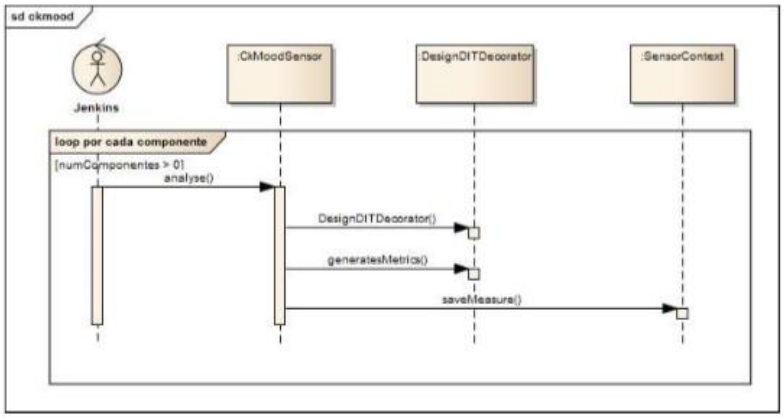
El siguiente diagrama muestra la aplicación del patrón Decorador para la generación de la métrica DIT

MetricasCKMOOD
Diseño Detallado



5.2 Escenario Básico

El siguiente escenario, muestra la generación de la métrica DIT para cada uno de los componentes de un proyecto java.



5.3 Escenarios Adicionales

Todas las demás métricas se generan de la misma forma, sólo cambia la clase Decorador de la métrica.

Casos de uso

Caso de Uso: CU1 - Obtener métricas

MetricasCKMOOD
Caso de Uso: CU1 - Obtener métricas

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
CU1 - Obtener métricas v1.0.pdf	1.0	24/03/2016	Ing. Ricardo Romero Villagómez	Borrador
CU1 - Obtener métricas v2.0.pdf	2.0	01/04/2016	Ing. Ricardo Romero Villagómez	Actualización
CU1 - Obtener métricas v3.0.pdf	3.0	08/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1 Descripción

Este caso de uso obtiene las métricas CK y MOOD para un proyecto java en específico.

2 Actores

Jenkins <<internal worker>>

3 Precondiciones

El *plugin* CKMOOD se encuentre instalado.

4 Flujo Básico de Eventos

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, ejecuta el análisis de calidad de un proyecto java en específico

Paso 2.- Obtiene las métricas CK por cada componente del proyecto java

Paso 3.- Obtiene las métricas MOOD por cada componente del proyecto java

Paso 4.- Presenta un resumen de las métricas obtenidas

Paso 5.- Fin del caso de uso

5 Flujos Alternativos

N/A

6 Subflujos

N/A

7 Escenarios Clave

N/A

8 Post-condiciones

1. Las métricas obtenidas por componente son almacenadas en la base de datos

9 Requerimientos Especiales

N/A

Caso de Uso: CU2 – Obtener métricas CK

MetricasCKMOOD
Caso de Uso: CU2 – Obtener métricas CK

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
CU2 – Obtener métricas CK v1.0.pdf	1.0	24/03/2016	Ing. Ricardo Romero Villagómez	Borrador
CU2 – Obtener métricas CK v2.0.pdf	2.0	01/04/2016	Ing. Ricardo Romero Villagómez	Actualización
CU2 – Obtener métricas CK v3.0.pdf	3.0	08/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1 Descripción

Este caso de uso obtiene las métricas CK para un proyecto java en específico. Las métricas en específico a obtener son: WMC (Weighted Methods Per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling Between Object Classes), RFC (Response for Class), LCOM (Lack of Cohesion of Methods), LCOM1 (Lack of Cohesion of Methods1), LCOM2 (Lack of Cohesion of Methods2), MPC (Message Passing Coupling), DAC (Data Abstraction Coupling) y SIZE2

2 Actores

Jenkins <<internal worker>>

3 Precondiciones

N/A

4 Flujo Básico de Eventos

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, ejecuta el análisis de calidad de un proyecto java en específico y por cada componente de dicho proyecto realizará los siguientes pasos.

Paso 2.- Obtener WMC del componente: La fórmula es $WMC=\sum_{i=1}^{n} c_i$, donde se obtienen el número de métodos de cada componente.

Paso 3.- Se almacena la métrica WMC obtenida del componente.

Paso 4.- Obtener DIT del componente: Esta métrica es la profundidad en el árbol de herencia de objetos.

Paso 5.- Se almacena la métrica DIT obtenida del componente.

Paso 6.- Obtener NOC del componente: El NOC de una clase es el número de subclases subordinadas en una jerarquía de objetos.

Paso 7.- Se almacena la métrica NOC obtenida del componente.

Paso 8.- Obtener CBO del componente: El CBO de una clase, es el número de clases con la que está acoplada, esto es, si usa sus métodos o variables de instancia.

Paso 9.- Se almacena la métrica CBO obtenida del componente.

Paso 10.- Obtener RFC del componente: El RFC de una clase, es el conjunto de métodos que se pueden ejecutar como respuesta de un mensaje recibido por un objeto. Está dado por la formula $RFC=|RS|$, donde $RS=\{M\}\cup_{all\,i}\{R_i\}$ y $\{R_i\}$ es el conjunto de métodos invocados por el método i y $\{M\}$ es el conjunto de todos los métodos de la clase.

Paso 11.- Se almacena la métrica RFC obtenida del componente.

Paso 12.- Obtener LCOM del componente: LCOM está definido por Sea $\{I_i\}$ el conjunto de variables de instancia usados por el método M_i, existen n de esos conjuntos $\{I_1, I_2, \ldots, I_n\}$, tal que $P=\{(I_i,I_j) \vee I_i \cap I_j = 0\}$ y $Q=\{(I_i,I_j) \vee I_i \cap I_j \neq 0\}$ entonces $LCOM = |P| - |Q|, si |P| > |Q|; de otra forma LCOM = 0.$

Paso 13.- Se almacena la métrica LCOM obtenida del componente.

Paso 14.- Obtener LCOM1 del componente: Esta métrica es el número de conjuntos disjuntos de métodos locales en una clase.

Paso 15.- Se almacena la métrica LCOM1 obtenida del componente.

Paso 16.- Obtener MPC del componente: La métrica MPC es el número de métodos invocados en una clase.

Paso 17.- Se almacena la métrica MPC obtenida del componente.

Paso 18.- Obtener DAC del componente: El DAC de un clase es el número de atributos de una clase que tiene como tipo a otra clase.

Paso 19.- Se almacena la métrica DAC obtenida del componente.

Paso 20.- Obtener SIZE2 del componente: Esta métrica es el número de atributos más el número de métodos de una clase.

Paso 21.- Se almacena la métrica SIZE2 obtenida del componente.

Paso 22.- Obtener LCOM2 del componente: Esta variante de la métrica LCOM se obtiene por medio de la siguiente formula: LCOM2 = (a – kl) / (l – kl), donde l = Número de atributos; k = Número de métodos y a = Sumatoria de los distintos atributo accedidos por cada método.

Paso 23.- Se almacena la métrica LCOM2 obtenida del componente.

Paso 24.- Fin del caso de uso

5 Flujos Alternativos

N/A

6 Subflujos

N/A

7 Escenarios Clave

N/A

8 Post-condiciones

N/A

9 Requerimientos Especiales

N/A

Caso de Uso: CU3 – Obtener métricas MOOD

MetricasCKMOOD
Caso de Uso: CU3 – Obtener métricas MOOD

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
CU3 – Obtener métricas MOOD v1.0.pdf	1.0	24/03/2016	Ing. Ricardo Romero Villagómez	Borrador
CU3 – Obtener métricas MOOD v2.0.pdf	2.0	01/04/2016	Ing. Ricardo Romero Villagómez	Actualización
CU3 – Obtener métricas MOOD v3.0.pdf	3.0	08/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1 Descripción

Este caso de uso obtiene las métricas MOOD para un proyecto java en específico. Las métricas en específico a obtener son: Polymorphism Factor (PF), Coupling Factor (CF), Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF)

2 Actores

Jenkins <<internal worker>>

3 Precondiciones

N/A

4 Flujo Básico de Eventos

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, ejecuta el análisis de calidad de un proyecto java en específico y por cada componente de dicho proyecto realizará los siguientes pasos.

Paso 2.- Obtener PF del componente: La fórmula es $PF=\frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i)*DC(C_i)]}$, Donde:

$M_o(C_i) = M_n(C_i) + M_o(C_i)$, $M_n(C_i)$ = Número de métodos definidos en la clase C_i, $M_o(C_i)$ = Número de métodos sobrecargados de la clase C_i, $DC(C_i)$ = Número de descendientes (hijos) de la clase C_i y TC = Total de clases

Paso 3.- Se almacena la métrica PF obtenida del componente.

Paso 4.- Obtener CF del componente: Esta métrica se obtiene por $CF=\frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is_{client}(C_I,C_J)\right]}{TC}$, donde $is_{client}(C_i,C_j)$ es la relación entre la clase cliente y la clase objeto y TC = total de clases

Paso 5.- Se almacena la métrica CF obtenida del componente.

Paso 6.- Obtener MHF del componente: Se obtiene por medio de la formula $MHF=\frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$, donde $M_d(C_i) = M_v(C_i) + M_h(C_i)$, $M_d(C_i)$ = Número de métodos definidos en la clase C_i , $M_v(C_i)$ = Número de métodos visibles de la clase C_i, $M_h(C_i)$ = Número de métodos ocultos de la clase C_i y TC = Total de clases

Paso 7.- Se almacena la métrica MHF obtenida del componente.

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Paso 8.- Obtener AHF del componente: El AHF se obtiene por , donde $A_d(C_i) = A_v(C_i) + A_h(C_i)$, $A_d(C_i)$ = Número de atributos definidos en la clase C_i , $A_v(C_i)$ = Número de atributos visibles de la clase C_i, $A_h(C_i)$ = Número de atributos ocultos de la clase C_i y TC = Total de clases

Paso 9.- Se almacena la métrica AHF obtenida del componente.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Paso 10.- Obtener MIF del componente: Se obtiene por la formula , donde $M_a(C_i) = M_d(C_i) + M_i(C_i)$, $M_a(C_i)$ = Número de métodos definidos en la clase C_i , $M_d(C_i)$ = Número de métodos declarados de la clase C_i, $M_i(C_i)$ = Número de métodos heredados de la clase C_i y TC = Total de clases

Paso 11.- Se almacena la métrica MIF obtenida del componente.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Paso 12.- Obtener AIF del componente: Para ésta métrica se aplica la formula , donde $A_a(C_i) = A_d(C_i) + A_i(C_i)$, $A_a(C_i)$ = Número de atributos definidos en la clase C_i , $A_d(C_i)$ = Número de atributos declarados de la clase C_i , $A_i(C_i)$ = Número de atributos heredados de la clase C_i y TC = Total de clases

Paso 13.- Se almacena la métrica AIF obtenida del componente.

Paso 14.- Fin del caso de uso

5 Flujos Alternativos

N/A

6 Subflujos

N/A

7 Escenarios Clave

N/A

8 Post-condiciones

N/A

9 Requerimientos Especiales

N/A

MetricasCKMOOD
Caso de Uso: CU4 – Exportar métricas

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
CU4 – Exportar métricas v1.0.pdf	1.0	15/04/2016	Ing. Ricardo Romero Villagómez	Borrador
CU4 – Exportar métricas v2.0.pdf	2.0	30/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1 Descripción

Este caso de uso permite exportar las métricas obtenidas después de realizar el análisis de un proyecto en Java

2 Actores

User

3 Precondiciones

N/A

4 Flujo Básico de Eventos

Paso 1.- Este caso de uso comienza cuando el actor Jenkins, después de ejecutar el análisis de calidad de un proyecto java en específico realizará los siguientes pasos.

Paso 2.- Selecciona el proyecto analizado

Paso 3.- Selecciona el reporte en formato PDF

Paso 4.- Descarga el archivo del reporte en formato PDF

Paso 5.- Fin del caso de uso

5 Flujos Alternativos

5.1 Flujo Alternativo 1

Paso 3.- Selecciona el reporte en formato CSV

Paso 4.- Descarga el archivo del reporte en formato CSV

Paso 5.- Fin del caso de uso

5.2 Flujo Alternativo 2

Paso 3.- Selecciona el reporte en formato XLS

Paso 4.- Descarga el archivo del reporte en formato XLS

Paso 5.- Fin del caso de uso

6 Subflujos

N/A

7 Escenarios Clave

N/A

8 Post-condiciones

1.- El reporte en formato PDF se creó.

2.- El reporte en formato CSV se creó.

3.- El reporte en formato XLS se creó.

9 Requerimientos Especiales

N/A

Caso de Uso: CU5 – Configurar métricas

MetricasCKMOOD
Caso de Uso: CU5 – Configurar métricas

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
CU5 – Configurar métricas v1.0.pdf	1.0	15/04/2016	Ing. Ricardo Romero Villagómez	Borrador
CU5 – Configurar métricas v2.0.pdf	2.0	30/04/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1 Descripción

Este caso de uso permite configurar la aplicación de la generación de las métricas CK y MOOD para un proyecto java en específico.

2 Actores

Use

3 Precondiciones

N/A

4 Flujo Básico de Eventos

Paso 1.- Este caso de uso comienza cuando el actor User, seleccionar configurar la generación de métricas.

Paso 2.- El actor selecciona desactivar la generación de métricas

Paso 3.- Fin del caso de uso

5 Flujos Alternativos

5.1 Flujo Alternativo 1

Paso 2.- El actor selecciona activar la generación de métricas

Paso 3.- Fin del caso de uso

6 Subflujos

N/A

7 Escenarios Clave

N/A

8 Post-condiciones

8.1 Post-condición 1

Para la siguiente ejecución del análisis de calidad para cualquier proyecto, no se generarán las métricas

8.2 Post-condición 2

Para la siguiente ejecución del análisis de calidad para cualquier proyecto, se generarán nuevamente las métricas

9 Requerimientos Especiales

N/A

Plan de pruebas

MetricasCKMOOD
Plan de Pruebas

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Plan de pruebas v1.0.pdf	1.0	21/05/2016	Ing. Ricardo Romero Villagómez	Borrador
Plan de pruebas v2.0.pdf	2.0	14/06/2016	Ing. Ricardo Romero Villagómez	Actualización
Plan de pruebas v3.0.pdf	3.0	14/07/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

1 Introducción

Este documento denominado Plan de Pruebas (PP) describe el alcance, estrategia general, recursos y calendario de las actividades de pruebas que se realizarán en el proyecto MetricasCKMOOD.

El PP identifica los elementos de prueba, rasgos o funcionalidad que será probada, tareas de pruebas, responsables de realizarlas y cualquier riesgo que requiera un plan de contingencia. Adicionalmente el PP guía a los ingenieros de software en la identificación y generación de casos de prueba de forma clara y concreta. De esta forma se asegura que todos los requerimientos del proyecto sean cubiertos y validados.

Este documento será la base para la verificación del funcionamiento correcto del proyecto proporcionando criterios técnicos y de operatividad para las pruebas unitarias, de integración y de sistema que apoyarán para obtener un sistema de calidad.

2 Propósito

Proporcionar un plan de pruebas general así como los criterios y casos de pruebas que logren asegurar, en su totalidad, el correcto desempeño y funcionalidad de cada uno de los componentes, verificando que se estén cubriendo los requerimientos para el proyecto, utilizando para ello desde criterios simples y rutinarios, hasta los casos más complejos y de menor incidencia.

3 Alcances

El PP para el proyecto proporcionará criterios tanto de operación como técnicos.

El PP considerará cada uno de los componentes del proyecto como elementos para las pruebas unitarias, pruebas de integración y pruebas finales de sistema que se realizaran en las instalaciones y con la infraestructura del programador.

La metodología de pruebas y este documento de PP permitirán al equipo de profesionales que participan en las pruebas del proyecto, evaluar aspectos como: la lógica estructural, la seguridad, la interconexión, el soporte conceptual, las herramientas de apoyo y sobretodo la independencia de aspectos técnicos del desarrollo de la solución tecnológica contratada, tales como: la plataforma tecnológica o la arquitectura de la solución a probar.

4 Definiciones, Acrónimos y Abreviaturas

- **Plan de prueba**: describe todos los métodos que se utilizarán para verificar que el software satisface la especificación del producto y las necesidades del cliente.

- **Casos de prueba**: lista los ítems específicos que serán probados y describe los pasos detallados que serán seguidos para verificar el software.
- **Reporte de pruebas**: describen los problemas encontrados al ejecutar los casos de prueba.
- **Herramientas de pruebas y automatización**: documentación de las herramientas empleadas en el proceso de pruebas.
- **Métricas, estadísticas y resúmenes**: indican como ha sido el progreso del proceso de prueba.

5 Estrategia General de Pruebas

En el proyecto se realizarán pruebas incrementales, es decir, en las pruebas unitarias evaluaremos casos propios de la funcionalidad de cada componente, en las pruebas de integración verificaremos únicamente la interacción con otros componentes del sistema y en las pruebas de sistema verificaremos el flujo completo que lleva a cabo el sistema.

5.1 Herramientas para Pruebas

Las herramientas que se utilizarán, dependerán del tipo de prueba que se realizará, es decir que por cada tipo de prueba es posible que se utilice una herramienta diferente.

5.1.1 JUnit

Características	Tipo de prueba
JUnit es un framework (conjunto de clases) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente. JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con	Esta herramienta será utilizada para la ejecución de: • Pruebas Unitarias. • Pruebas de Integración. • Pruebas de Sistema. • Pruebas de Regresión.

los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación. La versión de la herramienta que se utilizará es 4.11.	

5.1.2 Jenkins

Características	Tipo de prueba
Jenkins es un software de Integración continua escrito en Java. Jenkins proporciona integración continua para el desarrollo de software. La versión de la herramienta que se utilizará es 1.596	Esta herramienta será utilizada para la creación de scripts.

5.1.3 SonarQube

Características	Tipo de prueba
Conocido anteriormente como "Sonar", es una herramienta para evaluar código fuente. Es software libre y usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que ayuden a mejorar la calidad del código fuente de un programa. La versión de la herramienta que se utilizará es 4.5.2	Esta herramienta será utilizada para evaluar la calidad del código fuente.

5.2 Criterios de Aprobación / Falla

Los criterios de aprobación o falla del proyecto son los siguientes:

- Se deberán aplicar todos los casos de pruebas unitarias, integración, sistema y regresión definidos en este documento y sus resultados deberán ser similares a los resultados esperados en cada uno de éstos.
- Cubrir correctamente con todos los requerimientos del proyecto de acuerdo con las métricas definidas para cada requerimiento y a las pruebas realizadas.
- No deberá quedar ningún defecto detectado como pendiente al finalizar las pruebas.

Un componente es liberado cuando

- Cumple con los estándares de calidad.
- Cumple con los requerimientos definidos en el documento Requerimientos del Proyecto.doc

Un componente es rechazado cuando
- No cumple con los estándares de calidad.
- No cumple con los requerimientos definidos en el documento Requerimientos del Proyecto.doc

6 Tipos de Pruebas

En esta sección se mencionan de manera general los tipos de pruebas a realizar.

El objetivo principal de la ejecución de las pruebas esta dado para:

- Descubrir tantos errores como sea posible.
- Notificar acerca de los riesgos percibidos del proyecto.
- Identificar errores funcionales de la aplicación, enmarcadas en grados de usabilidad ya definidos.
- Evaluar la calidad del producto y señalar un indicador de aceptación del mismo.
- Evaluar la calidad técnica del producto y resolver los errores identificados en las pruebas de tipo técnico.
- Cumplir con los requerimientos específicos del cliente, en cuanto a la ejecución de las pruebas.

6.1 Pruebas Unitarias

Las pruebas unitarias tienen como objetivo verificar la funcionalidad y estructura de cada componente individualmente del sistema una vez que ha sido codificado.

Es una Prueba técnica que permitirá:

- Verificar que los módulos del sistema estén libres de errores.
- Que todos los caminos lógicos principales deben ejecutarse correctamente en cada módulo de la aplicación.
- Todas las transacciones deben ser probadas.
- Todos los tipos de registro de entrada válidos deben ser procesados.

- Todos los tipos de registro de entrada inválidos deben ser procesados correctamente.
- Excepciones a tratamiento normal.
- Todas las salidas válidas son procesadas.

6.2 Pruebas de Integración

El objetivo de las pruebas de integración es verificar el correcto ensamblaje entre los distintos módulos que componen la solución una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces internas y externas, que cubren la funcionalidad establecida.

En esta prueba se comprueba la compatibilidad y funcionalidad de los interfaces entre las distintas "partes" que componen el desarrollo de la solución. Estas partes pueden ser módulos, aplicaciones individuales, es decir esta prueba válida la integración entre los diferentes módulos que componen la solución con el fin de garantizar que su operación integrada es correcta, teniendo en cuenta los siguientes temas técnicos:

- El funcionamiento integrado de módulos interdependientes debe estar libre de errores.
- Probar todas las dependencias entre módulos.
- Probar el flujo de control y el flujo de datos a través de todas las capas.

6.3 Pruebas de Sistemas

Las pruebas de sistema buscan diferencias entre la solución desarrollada y los requerimientos, enfocándose en la identificación de los errores que se puedan generar entre la especificación funcional y el diseño del sistema, así como, el negocio objeto de la aplicación.

6.4 Pruebas de Regresión

En esta prueba se valida que el sistema mantenga su correcta funcionalidad después de la incorporación de un ajuste, corrección o nuevo requerimiento. Es una prueba funcional y técnica que valida que el sistema siga funcionando perfectamente después de que las correcciones sean aplicadas.

7 Documentos de Pruebas a Entregar

Los siguientes documentos entregables forman parte del plan de pruebas:

- Plan de pruebas.
- Descripción de casos de pruebas para las fases de pruebas unitarias.

8 Actividades de Pruebas

Las actividades que se realizarán en la preparación y ejecución de pruebas por cada fase serán las siguientes:

- Configurar el ambiente de pruebas.
- Analizar los componentes a probar.
- Definir los casos de prueba de acuerdo al estándar de casos de prueba.
- Crear los casos de prueba para todas las fases de pruebas definidas para el proyecto.
- Ejecución de pruebas.

8.1 Pruebas Unitarias

- Configurar el ambiente de pruebas unitarias por parte del desarrollador del componente.
- El desarrollador del componente obtendrá datos reales para las pruebas unitarias de lo contrario preparará datos ficticios cercanos a la realidad.
- Para las pruebas manuales, ejecutar cada caso de prueba unitaria por componente según corresponda.
- Para las pruebas manuales, en el caso de que se detecten defectos durante la ejecución, el dueño del componente llevará a cabo la corrección de defectos.

En caso de defectos:

- En caso de utilizar la herramienta de integración continua, notificará a las personas indicadas en caso de fallo para las pruebas de unidad que se ejecutan de forma automática.
- El dueño del componente llevará a cabo la corrección de defectos y en la siguiente construcción del producto, se volverá a llevar a cabo la ejecución automática de las pruebas.

8.2 Construcción del Producto

- Se realizará la integración de todos los componentes del proyecto antes de llevar a cabo las pruebas de integración y de sistema.

- El responsable de pruebas deberá asegurar que todos los componentes han sido probados de manera individual y que los resultados han sido satisfactorios.
- El responsable de calidad se asegurará que los componentes a integrarse ha cumplido con los objetivos de calidad estipulados.
- Cualquier componente que no cumpla con la calidad especificada o que las pruebas unitarias no tengan resultados correctos no podrá ser integrado al producto.
- La integración de los componentes se llevará a cabo de forma automática por medio de la herramienta de integración continua definida para el proyecto. Para el caso de los proyectos que utilicen plataformas que no estén configuradas en la herramienta de integración continua, el responsable de implementación será el que realice la construcción e integración del producto.
- El encargado de la construcción deberá:
    - Identificar los componentes a integrar.
    - Verificar las dependencias entre cada uno de los componentes.
    - Verificar la secuencia de integración más óptima.
    - Establecer el ambiente de integración según los requerimientos especificados.
    - Asegurar que todas las interfaces han sido identificadas, están completas y bien definidas.
    - Realizar la construcción e integración del producto.

8.3 Pruebas de Integración

- Asegurar que los componentes han sido integrados.
- Configurar el ambiente de pruebas de integración por parte del responsable de prueba y el responsable de soporte.
- El responsable de pruebas obtendrá datos reales para las pruebas de integración de lo contrario preparará datos ficticios cercanos a la realidad.
- El responsable de pruebas dará de alta los datos de prueba de integración en el ambiente configurado.
- El encargado o los encargados de ejecutar las pruebas de integración obtendrán todos los componentes del sistema con sus pruebas unitarias liberadas.
- Para las pruebas manuales, ejecutar cada caso de prueba unitaria por componente según corresponda.
- Para las pruebas manuales, en el caso de que se detecten defectos durante la ejecución darlo de alta, haciendo referencia al caso de prueba con el cual está relacionado y continuar con la ejecución de la prueba, la cual seguirá en estado “Prueba en Progreso”
- Para el caso de las pruebas manuales, el criterio para cerrar el caso de prueba es haber ejecutado todos los pasos definidos en la misma.

En caso de defectos:

- En caso de utilizar la herramienta de integración continua, notificará a las personas indicadas en caso de fallo para las pruebas de integración que se ejecutan de forma automática.
- Los dueños de los componentes llevarán a cabo la corrección de defectos y en la siguiente construcción del producto, se volverá a llevar a cabo la ejecución automática de las pruebas.
- Para el caso de las pruebas manuales, el líder de proyecto deberá asignar la corrección del defecto al dueño del componente, el cual será notificado por correo electrónico.

8.4 Pruebas de Sistema

- Configurar el ambiente de pruebas de sistema por parte del responsable de pruebas y el responsable de soporte.
- El responsable de pruebas deberá obtener datos reales para las pruebas de sistema de lo contrario preparar datos ficticios cercanos a la realidad.
- El responsable de pruebas dará de alta los datos de prueba de sistema en el ambiente configurado.
- El encargado o los encargados de ejecutar las pruebas de sistema obtendrán todos los componentes del sistema con sus pruebas de integración liberadas.
- El encargado o los encargados de las pruebas de sistema ejecutarán cada caso de prueba de sistema.
- Para las pruebas manuales, ejecutar cada caso de prueba unitaria por componente según corresponda.
- Para las pruebas manuales, en el caso de que se detecten defectos durante la ejecución darlo de alta, haciendo referencia al caso de prueba con el cual está relacionado y continuar con la ejecución de la prueba, la cual seguirá en estado “Prueba en Progreso”
- Para el caso de las pruebas manuales, el criterio para cerrar el caso de prueba es haber ejecutado todos los pasos definidos en la misma.

En caso de defectos:

- El líder del módulo deberá asignar la corrección del defecto a la persona correspondiente, la cual será notificada por correo electrónico.

Una vez cubiertos satisfactoriamente todos los requerimientos con los casos de pruebas se procederá a la liberación del proyecto.

9 Responsabilidades y Necesidades del Ambiente

La conducción y monitoreo de las pruebas unitarias, integración y sistema estarán a cargo del responsable de pruebas designado por el equipo de desarrollo del proyecto, sin embargo ni el responsable de pruebas ni los encargados de ejecutar las pruebas en cualquier fase del proyecto tendrán a su cargo la búsqueda de los defectos detectados en las pruebas realizadas.

Los ingenieros que estarán a cargo de cada una de las pruebas para los diferentes componentes del proyecto serán:

9.1 Pruebas Unitarias

Id	Componente	Responsable
PU-1	*Obtener Métricas*	*Ing. Ricardo Romero*
PU-2	*Obtener Métricas CK*	*Ing. Ricardo Romero*
PU-3	*Obtener Métricas MOOD*	*Ing. Ricardo Romero*
PU-5	*Exportar Métricas*	*Ing. Ricardo Romero*
PU-6	*Configurar Métricas*	*Ing. Ricardo Romero*

9.2 Pruebas de Integración

Id	Componente	Responsable
PI-1	*Obtener Métricas*	*Ing. Ricardo Romero*
PI-2	*Obtener Métricas CK*	*Ing. Ricardo Romero*
PI-3	*Obtener Métricas MOOD*	*Ing. Ricardo Romero*
PI-4	*Exportar Métricas*	*Ing. Ricardo Romero*
PI-5	*Configurar Métricas*	*Ing. Ricardo Romero*

9.3 Pruebas de Sistema

Id	Componente	Responsable
PS-1	*Obtener Métricas*	*Ing. Ricardo Romero*

Id	Componente	Responsable
PS-2	*Obtener Métricas CK*	*Ing. Ricardo Romero*
PS-3	*Obtener Métricas MOOD*	*Ing. Ricardo Romero*
PS-4	*Exportar Métricas*	*Ing. Ricardo Romero*
PS-5	*Configurar Métricas*	*Ing. Ricardo Romero*

10 Calendario Pruebas

Las siguientes tablas muestran las fechas estimadas para la terminación de cada una de las pruebas en el proyecto.

10.1 Pruebas Unitarias

Id	Componente	*Fecha Estimada de Inicio*	*Fecha Estimada de Termino*
PU-1	*Obtener Métricas*		
PU-2	*Obtener Métricas CK*		
PU-3	*Obtener Métricas MOOD*		
PU-4	*Exportar Métricas*		
PU-5	*Configurar Métricas*		

10.2 Pruebas de Integración

Id	Componente	*Fecha Estimada de Inicio*	*Fecha Estimada de Termino*
PI-1	*Obtener Métricas*		
PI-2	*Obtener Métricas CK*		
PI-3	*Obtener Métricas MOOD*		
PI-4	*Exportar Métricas*		
PI-5	*Configurar Métricas*		

10.3 Pruebas de Sistema

Id	Componente	*Fecha Estimada de Inicio*	*Fecha Estimada de Termino*
PS-1	*Obtener Métricas*		
PS-2	*Obtener Métricas CK*		
PS-3	*Obtener Métricas MOOD*		
PS-4	*Exportar Métricas*		
PS-5	*Configurar Métricas*		

11 Riesgos y Contingencias

Riesgo	*Contingencia*
El encargado de ejecutar las pruebas no se encuentra disponible para realizar las pruebas.	Otro miembro del equipo tomará el lugar del ingeniero que no pueda realizar las pruebas.
Existencia de defectos después de las pruebas no corregidos en la fase adecuada.	Monitoreo del estatus de las correcciones por el responsable de pruebas.
Configuración no disponible para realizar las pruebas.	Calendarizar y revisar con anticipación las fechas estimadas para pruebas para que la configuración se encuentre lista.

12 Aprobaciones

El plan de pruebas y todos los elementos que lo componen (Reportes de seguimientos de pruebas, casos de prueba para las diferentes fases, documentación correspondiente a pruebas) deberán ser aprobados en su totalidad por el líder de y monitoreados por el responsable de pruebas del proyecto MetricasCKMOOD donde el responsable de las pruebas es el Ing. Ricardo Romero Villagómez.

Casos de prueba

MetricasCKMOOD

Casos de Prueba

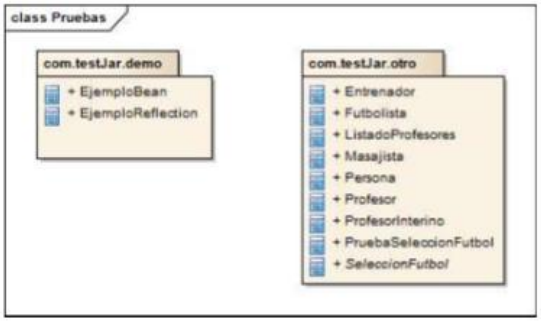
Control de Versiones

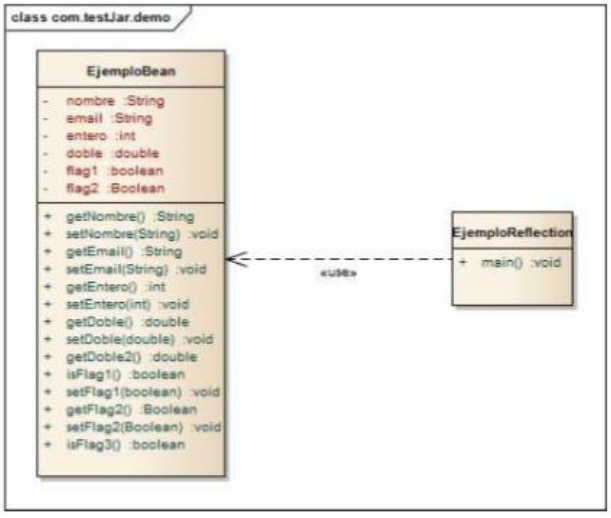
Nombre del archivo	Versión	Fecha	Autor	Comentarios
Casos de prueba v1.0.pdf	1.0	25/03/2016	Ing. Ricardo Romero Villagómez	Borrador
Casos de prueba v2.0.pdf	2.0	05/05/2016	Ing. Ricardo Romero Villagómez	Actualización
Casos de prueba v3.0.pdf	3.0	13/07/2016	Ing. Ricardo Romero Villagómez	Documento Final

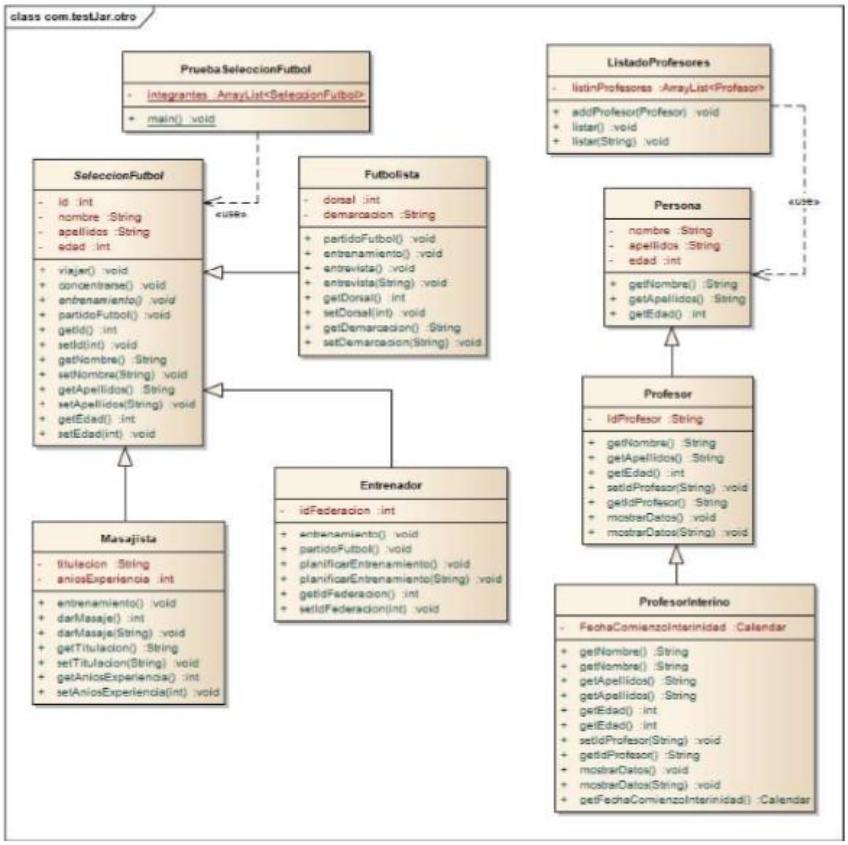
Contenido

PRE- CONDICIONES

Para realizar las pruebas unitarias de la generación de las métricas, se creó un archivo llamado testJar.jar, el cual será analizado antes de ejecutar cada prueba unitaria. Este archivo está compuesto por dos carpetas y por 11 clases java, con la siguiente estructura:






class com.testJar.otro
PruebaSeleccionFutbol
- integrantes :ArrayList<SeleccionFutbol>
+ main() :void
ListadoProfesores
- listinProfesores :ArrayList<Profesor>
+ addProfesor(Profesor) :void
+ listar() :void
+ listar(String) :void
SeleccionFutbol
- id :int
- nombre :String
- apellidos :String
- edad :int
+ viajar() :void
+ concentrarse() :void
+ entrenamiento() :void
+ partidoFutbol() :void
+ getId() :int
+ setId(int) :void
+ getNombre() :String
+ setNombre(String) :void
+ getApellidos() :String
+ setApellidos(String) :void
+ getEdad() :int
+ setEdad(int) :void
«use»
Futbolista
- dorsal :int
- demarcacion :String
+ partidoFutbol() :void
+ entrenamiento() :void
+ entrevista() :void
+ entrevista(String) :void
+ getDorsal() :int
+ setDorsal(int) :void
+ getDemarcacion() :String
+ setDemarcacion(String) :void
Persona
- nombre :String
- apellidos :String
- edad :int
+ getNombre() :String
+ getApellidos() :String
+ getEdad() :int
«use»
Profesor
- IdProfesor :String
+ getNombre() :String
+ getApellidos() :String
+ getEdad() :int
+ setIdProfesor(String) :void
+ getIdProfesor() :String
+ mostrarDatos() :void
+ mostrarDatos(String) :void
Entrenador
- idFederacion :int
+ entrenamiento() :void
+ partidoFutbol() :void
+ planificarEntrenamiento() :void
+ planificarEntrenamiento(String) :void
+ getIdFederacion() :int
+ setIdFederacion(int) :void
Masajista
- titulacion :String
- aniosExperiencia :int
+ entrenamiento() :void
+ darMasaje() :int
+ darMasaje(String) :void
+ getTitulacion() :String
+ setTitulacion(String) :void
+ getAniosExperiencia() :int
+ setAniosExperiencia(int) :void
ProfesorInterino
- FechaComienzoInterinidad :Calendar
+ getNombre() :String
+ getNombre() :String
+ getApellidos() :String
+ getApellidos() :String
+ getEdad() :int
+ getEdad() :int
+ setIdProfesor(String) :void
+ getIdProfesor() :String
+ mostrarDatos() :void
+ mostrarDatos(String) :void
+ getFechaComienzoInterinidad() :Calendar

TestCase01-testGetWmc:

Descripción: Este caso de prueba es para obtener la métrica WMC de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	14
com.testJar.demo.EjemploReflection	1
com.testJar.otro.Entrenador	6
com.testJar.otro.Futbolista	8
com.testJar.otro.ListadoProfesores	3
com.testJar.otro.Masajista	7
com.testJar.otro.Persona	3
com.testJar.otro.Profesor	4
com.testJar.otro.ProfesorInterino	3
com.testJar.otro.PruebaSeleccionFutbol	1
com.testJar.otro.SeleccionFutbol	12

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase02-testGetNoc:

Descripción: Este caso de prueba es para obtener la métrica NOC de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	0
com.testJar.demo.EjemploReflection	0
com.testJar.otro.Entrenador	0
com.testJar.otro.Futbolista	0
com.testJar.otro.ListadoProfesores	0
com.testJar.otro.Masajista	0
com.testJar.otro.Persona	1
com.testJar.otro.Profesor	1
com.testJar.otro.ProfesorInterino	0
com.testJar.otro.PruebaSeleccionFutbol	0
com.testJar.otro.SeleccionFutbol	3

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase03-testGetRfc:

Descripción: Este caso de prueba es para obtener la métrica RFC de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	14
com.testJar.demo.EjemploReflection	13
com.testJar.otro.Entrenador	9
com.testJar.otro.Futbolista	11
com.testJar.otro.ListadoProfesores	12
com.testJar.otro.Masajista	10
com.testJar.otro.Persona	3
com.testJar.otro.Profesor	10
com.testJar.otro.ProfesorInterino	9
com.testJar.otro.PruebaSeleccionFutbol	14
com.testJar.otro.SeleccionFutbol	13

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase04-testGetDit:

Descripción: Este caso de prueba es para obtener la métrica DIT de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	1
com.testJar.demo.EjemploReflection	1
com.testJar.otro.Entrenador	2
com.testJar.otro.Futbolista	2
com.testJar.otro.ListadoProfesores	1
com.testJar.otro.Masajista	2
com.testJar.otro.Persona	1
com.testJar.otro.Profesor	2
com.testJar.otro.ProfesorInterino	3
com.testJar.otro.PruebaSeleccionFutbol	1
com.testJar.otro.SeleccionFutbol	1

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase05-testGetCbo:

Descripción: Este caso de prueba es para obtener la métrica CBO de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	0
com.testJar.demo.EjemploReflection	0
com.testJar.otro.Entrenador	1
com.testJar.otro.Futbolista	1
com.testJar.otro.ListadoProfesores	0
com.testJar.otro.Masajista	1
com.testJar.otro.Persona	1
com.testJar.otro.Profesor	2
com.testJar.otro.ProfesorInterino	0
com.testJar.otro.PruebaSeleccionFutbol	0
com.testJar.otro.SeleccionFutbol	4

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase06-testGetLcom:

Descripción: Este caso de prueba es para obtener la métrica LCOM de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	2
com.testJar.demo.EjemploReflection	0
com.testJar.otro.Entrenador	0
com.testJar.otro.Futbolista	0
com.testJar.otro.ListadoProfesores	0
com.testJar.otro.Masajista	0
com.testJar.otro.Persona	2
com.testJar.otro.Profesor	2
com.testJar.otro.ProfesorInterino	0
com.testJar.otro.PruebaSeleccionFutbol	0
com.testJar.otro.SeleccionFutbol	0

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase07-testGetLcom1:

Descripción: Este caso de prueba es para obtener la métrica LCOM1 de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	2
com.testJar.demo.EjemploReflection	0
com.testJar.otro.Entrenador	2
com.testJar.otro.Futbolista	2
com.testJar.otro.ListadoProfesores	0
com.testJar.otro.Masajista	2
com.testJar.otro.Persona	2
com.testJar.otro.Profesor	2
com.testJar.otro.ProfesorInterino	0
com.testJar.otro.PruebaSeleccionFutbol	0
com.testJar.otro.SeleccionFutbol	2

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase08-testGetLcom2:

Descripción: Este caso de prueba es para obtener la métrica LCOM2 de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	0.9423
com.testJar.demo.EjemploReflection	2
com.testJar.otro.Entrenador	0
com.testJar.otro.Futbolista	0.5714
com.testJar.otro.ListadoProfesores	0
com.testJar.otro.Masajista	0.5833
com.testJar.otro.Persona	1
com.testJar.otro.Profesor	0
com.testJar.otro.ProfesorInterino	0
com.testJar.otro.PruebaSeleccionFutbol	0
com.testJar.otro.SeleccionFutbol	0.8182

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase09- testGetMpc:

Descripción: Este caso de prueba es para obtener la métrica MPC de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	4
com.testJar.demo.EjemploReflection	10
com.testJar.otro.Entrenador	7
com.testJar.otro.Futbolista	7
com.testJar.otro.ListadoProfesores	9
com.testJar.otro.Masajista	7
com.testJar.otro.Persona	3
com.testJar.otro.Profesor	7
com.testJar.otro.ProfesorInterino	9
com.testJar.otro.PruebaSeleccionFutbol	12
com.testJar.otro.SeleccionFutbol	5

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase10-testGetDac:

Descripción: Este caso de prueba es para obtener la métrica DAC de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	8
com.testJar.demo.EjemploReflection	0
com.testJar.otro.Entrenador	1
com.testJar.otro.Futbolista	2
com.testJar.otro.ListadoProfesores	1
com.testJar.otro.Masajista	2
com.testJar.otro.Persona	3
com.testJar.otro.Profesor	1
com.testJar.otro.ProfesorInterino	1
com.testJar.otro.PruebaSeleccionFutbol	1
com.testJar.otro.SeleccionFutbol	4

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase11-testGetSize2:

Descripción: Este caso de prueba es para obtener la métrica NOM de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
com.testJar.demo.EjemploBean	22
com.testJar.demo.EjemploReflection	1
com.testJar.otro.Entrenador	7
com.testJar.otro.Futbolista	10
com.testJar.otro.ListadoProfesores	4
com.testJar.otro.Masajista	9
com.testJar.otro.Persona	6
com.testJar.otro.Profesor	5
com.testJar.otro.ProfesorInterino	4
com.testJar.otro.PruebaSeleccionFutbol	2
com.testJar.otro.SeleccionFutbol	16

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase12-testGetPf:

Descripción: Este caso de prueba es para obtener la métrica PF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0.9354

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase13-testGetCf:

Descripción: Este caso de prueba es para obtener la métrica CF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0.75

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase14-testGetMhf:

Descripción: Este caso de prueba es para obtener la métrica MHF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada

Datos requeridos: Archivo testJar.jar

TestCase15-testGetAhf:

Descripción: Este caso de prueba es para obtener la métrica AHF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0.3333

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase16-testGetMif:

Descripción: Este caso de prueba es para obtener la métrica MIF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0.6760

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

TestCase17-testGetAif:

Descripción: Este caso de prueba es para obtener la métrica AIF de las clases contenidas en el archivo testJar.jar. Los valores esperados son:

Archivo	Valor de la métrica
C:\Users\RicardoRomero\Downloads\testJar.jar	0

Pre-condiciones: Ejecución del análisis del archivo testJar.jar
Post-condiciones: La métrica ha sido calculada
Datos requeridos: Archivo testJar.jar

Código fuente

NOTA: Debido a la gran cantidad de líneas de código, el código fuente no formará parte del apéndice, y estará disponible en el disco compacto (CD) que será entregado junto con ésta investigación.

APÉNDICE 4. ENTREGABLES FASE TRANSICIÓN

Los documentos que componen esta fase son:

Manual de instalación

Documentación técnica

Manual de instalación

MetricasCKMOOD

Manual de Instalación

Control de Versiones

Nombre del archivo	Versión	Fecha	Autor	Comentarios
Manual de Instalación v1.0.pdf	1.0	10/10/2016	Ing. Ricardo Romero Villagómez	Borrador
Manual de Instalación v2.0.pdf	2.0	18/10/2016	Ing. Ricardo Romero Villagómez	Documento Final

Contenido

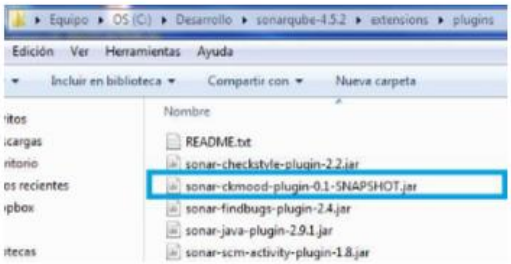
INSTALACIÓN

La instalación del componente MetricasCKMOOD es muy simple. Para llevar a cabo la instalación del componente MetricasCKMOOD, se necesitan seguir los siguientes pasos:

1. Detener la herramienta SonarQube.
2. Copiar el archivo del componente (sonar-ckmood-plugin-0.1-SNAPSHOT.jar) en la carpeta {directorio de instalación de la herramienta SonarQube}\extensions\plugins
3. Levantar la herramienta SonarQube.

Una vez que la herramienta SonarQube se termine de levantar, el componente estará instalado.

La siguiente imagen, muestra la carpeta en donde se deberá de copiar el archivo de componente, tomando como referencia que el equipo donde está instalada la herramienta SonarQube es Windows.



INTERFAZ GRAFICA

Después de realizar la instalación del componente, debemos de hacer visible la interfaz gráfica del componente.

1. Iniciamos sesión en la herramienta SonarQube, seleccionando la opción "Log in"
2. Seleccionamos un proyecto analizado
3. Seleccionamos la opción "Configure Widgets"

4. Buscamos el "CKMOOD report widget"

5. Seleccionamos la opción "Add widget"
6. Finalmente seleccionamos la opción "Back to dashboard" para salir de la configuración de widgets

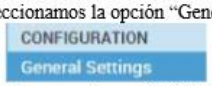
ACTIVACION/DESACTIVACION

El componente MetricasCKMOOD una vez instalado se encuentra en modo activado, esto quiere decir que para todos los análisis que se realicen, se ejecutará el análisis de las métricas CK – MOOD.

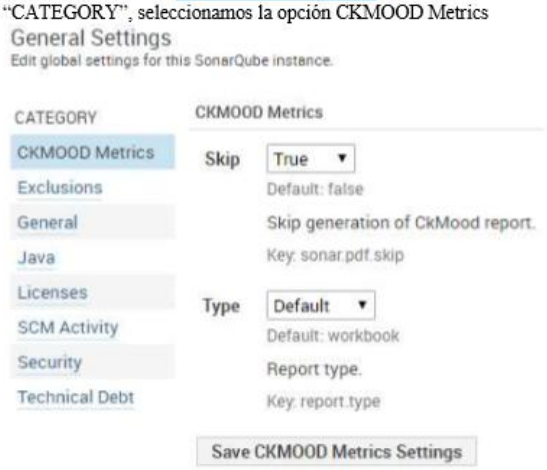
Los pasos para activar o desactivar el análisis de las métricas CK – MOOD son:

1. Iniciamos sesión en la herramienta SonarQube, seleccionando la opción "Log in"
2. Seleccionamos la opción "Settings"

3. Del menú de configuración seleccionamos la opción "General Metrics"



4. Del menú "CATEGORY", seleccionamos la opción CKMOOD Metrics



5. En la opción "Skip", es donde se activa o desactiva el análisis de las métricas
6. Después de hacer los cambios en la opción "Skip", se selecciona la opción "Save CKMOOD Metrics Settings"
7. Finalmente, seleccionamos la opción "Dashboards" para regresar a la pantalla principal

Documentación técnica

NOTA: Debido a la gran tamaño de la documentación, éste artefacto no formará parte del apéndice, y estará disponible en el disco compacto (CD) que será entregado junto con ésta investigación.

REFERENCIAS

Abu Asad, A. & Izzat, A., 2014. Evaluating the impact of software metrics on defects prediction. Part 2. *Computer Science Journal of Moldova*, 22(64), pp.127–144.

Arapidis, C.S., 2012. *Sonar Code Quality Testing Essentials* First., Inglaterra: Packt Publishing.

Badri, L., Badri, M. & Toure, F., 2011. An Empirical Analysis of Lack of Cohesion Metrics for Predicting Testability of Classes. *International Journal of Software Engineering and Its Applications*, 5(2), pp.69–86.

Berg, A.M., 2015. *Jenkins Continuous Integration Cookbook* Second., Inglaterra: Packt Publishing.

Brito, F., 2005. Design Metrics for Object-Oriented Software Systems. *Workshop on Quantitative Methods for Object-Oriented Systems Development*.

Campbell, A. & Papapetrou, P.P., 2014. *SonarQube in Action* First., Estados Unidos: Manning.

CastSoftware, 2015. CastSoftware. *Application Intelligence Platform*. Available at: http://www.castsoftware.com/products/application-intelligence-platform [Accessed November 25, 2015].

Chacon, S., 2015. Pro Git, el libro oficial de Git. Available at: http://librosweb.es/libro/pro_git/ [Accessed October 25, 2015].

Cheikhi, L. et al., 2014. Chidamber and Kemerer Object-Oriented Measures: Analysis of their Design from the Metrology Perspective. *International Journal of Software Engineering and Its Applications*, 8(2), pp.359–374.

Crispin, L. & Gregory, J., 2009. *Agile Testing: A practical guide for testers and agile teams* First., Estados Unidos: Addison Wesley.

Deshpande, A. & Riehle, D., 2008. Continuous Integration in Open Source Software Development, Fourth Conference on Open Source Systems. In *Fourth Conference on Open Source Systems*. Springer Verlag, pp. 273–280.

Duvall, P.M., Matyas, S. & Glover, A., 2007. *Continuous Integration Improving Software Quality and Reducing Risk*, Estados Unidos: Addison-Wesley.

Eclipse Foundation, 2012. Eclipse Process Framework (EPF). *OpenUP*. Available at: http://epf.eclipse.org/wikis/openup/ [Accessed November 25, 2015].

Expósito, R., 2009. ¿Qué es el Análisis Estático del Código ? , pp.1–19. Available at: http://raulexposito.com/documentos/analisis-estatico-codigo/ [Accessed November 4, 2015].

Ferguson, J., 2011. *Jenkins: The Definitive Guide* First., Estados Unidos: O'Reilly.

Fowler, M., 2015. Continuous Integration. *Continuous Integration*. Available at: http://www.martinfowler.com/articles/continuousIntegration.html [Accessed October 23, 2015].

Gigleux, A. & Campbell, A., 2016. SonarQube Documentation. *SonarQube Documentation*. Available at: http://docs.sonarqube.org/display/HOME/SonarQube+Platform [Accessed March 7, 2016].

Gorton, I., 2011. *Essential Software Architecture* Second., Estados Unidos: Springer.

Greiner, C. et al., 2010. Una propuesta de solución para automatizar la medición de aplicaciones orientadas a objeto. *XVI CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN*, pp.654–663.

Herbold, S., Waack, S. & Grabowski, J., 2011. Calculation and optimization of thresholds for sets of software metrics. *Empir Software Eng*, 16, pp.812–841.

Holck, J. & Jørgensen, N., 2007. Continuous integration and quality assurance: A case study of two open source projects. *Australasian Journal of Information Systems*, (2003), pp.40–53. Available at: http://dl.acs.org.au/index.php/ajis/article/viewArticle/145.

Holck, J. & Jørgensen, N., 2004. Continuous Integration as a Means of Coordination. *Constructing the Infrastructure for the Knowledge Economy*, pp.187–198.

HP, 2015. HP. *static-code-analysis-sast*. Available at: http://www8.hp.com/mx/es/software-solutions/static-code-analysis-sast [Accessed November 25, 2015].

Humble, J. & Farley, D., 2011. *Continuous Delivery*, Estados Unidos: Addison-Wesley.

IBM, 2015. IBM. *IBM Rational AppScan Source Edition*. Available at: http://www-01.ibm.com/support/knowledgecenter/SSS9LM_8.5.0/com.ibm.security.appscansrc.infocenter.nav.doc/helpindex.html?lang=es [Accessed November 25, 2015].

ISO, 2015. Online Browsing Platform (OBP). *ISO - International Organization for Standardization*.

Katsubo, D., 2015. CI feature matrix. *CI feature matrix*. Available at: https://www.centurion.link/w/_media/software/ci_feature_matrix.pdf [Accessed October 23, 2015].

Kiuwan, 2015. Kiuwan. *Kiuwan Software*. Available at: https://www.kiuwan.com [Accessed November 11, 2015].

Lamas, I., 2011. *Comparación de analizadores estáticos para código java*, España.

Li, W. & Henry, S., 1993. *Object Oriented Metrics Wich Predict Maintainability*, Estados Unidos.

Lluna, E., 2011. Análisis estático de código en el ciclo de desarrollo de software de seguridad crítica. *Revista Española de Innovación, Calidad e Ingeniería del Software*, 7(3), pp.26–38.

Loeliger, J., 2009. *Version Control with Git* First., Estados Unidos: O'REILLY.

Ma, Y.-T. et al., 2010. A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, 25(6), pp.1184–1201.

Meyer, B., 1999. *Construcción de Software Orientado a Objetos* Segunda., España: Prentice Hall.

Microsoft, 2015. Microsoft. *MSDN*. Available at: https://msdn.microsoft.com/en-us/library/ms194922(v=vs.90).aspx [Accessed November 11, 2015].

Miranda, E.A., Berón, M.M. & Riesco, D.E., 2014. Extracción y análisis de información estática

orientada a la comprensión de programas para Sistemas OO. *Ciencia y Tecnología*, 14, pp.163–182.

Miravet, P. et al., 2011. ANÁLISIS Y EVALUACIÓN DE HERRAMIENTAS DE CONTROL DE VERSIONES EN PROYECTOS SOFTWARE. *XV Congreso Internacional de Ingeniería de Proyectos*, pp.2390–2405.

Misra, S. & Bhavsar, V., 2003. Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. *Springer-Verlag*, pp.724–732.

Olmedilla, J.J., 2005. *Revisión Sistemática de Métricas de Diseño Orientado a Objetos*, España.

Pressman, R.S., 2010. *Ingeniería de software -un enfoque práctico-* Séptima., México: Mc Graw Hill.

Rehn, C., 2015. Continuous Integration: Aspects in Automation and Configuration Management. Available at: http://www.christian-rehn.de/wp-content/uploads/downloads/2012/04/seminar_ci.pdf.

Ritchie, S., 2011. *Pro .NET Best Practices*, Estados Unidos: Apress.

Roberts, M., 2004. Enterprise Continuous Integration Using Binary Dependencies. In *5th International Conference, XP 2004*. Alemania: Springer Verlag, pp. 194 – 201.

Rogers, O., 2004. Scaling Continuous Integration. *Extreme Programming and Agile Processes in Software Engineering*, 3092, pp.68 – 76.

Rutar, N., Almazan, C.B. & Foster, J.S., 2004. A Comparison of Bug Finding Tools for Java. *15th International Symposium on Software Reliability Engineering*, pp.245 – 256.

Schach, S., 2005. *Object orientated and classical software engineering* Eighth., Estados Unidos: Mc Graw Hill.

Singh, S. & Kahlon, K.S., 2014. Object oriented software metrics threshold values at quantitative acceptable risk level. *CSIT*, 2(3), pp.191–205.

Stansberry, G., 2008. 7 Version Control Systems Reviewed. *Smashing Magazine*, 09. Available at: http://www.smashingmagazine.com/2008/09/the-top-7-open-source-version-control-systems/.

Wang, H., Khoshgoftaar, T. & Liang, Q., 2013. A STUDY OF SOFTWARE METRIC SELECTION TECHNIQUES: STABILITY ANALYSIS AND DEFECT PREDICTION MODEL PERFORMANCE. *International Journal on Artificial Intelligence Tools*, 22(5), p.1360010 1 –1360010 25.