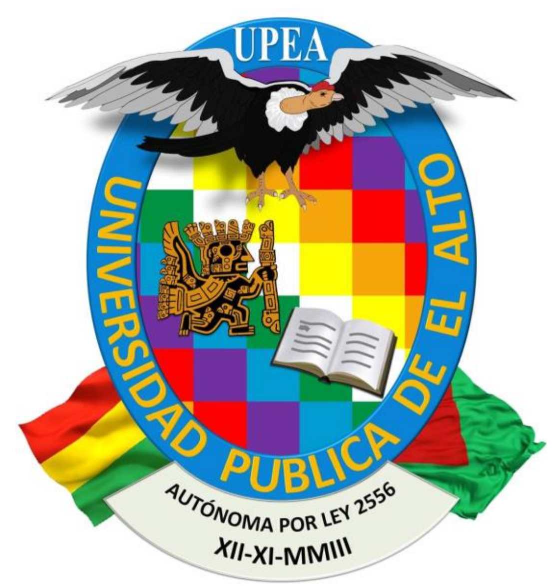
UNIVERSIDAD PÚBLICA DE EL ALTO

CARRERA INGENIERÍA DE SISTEMAS



TESIS DE GRADO

TÉCNICA DE MEDICIÓN DE CONFIABILIDAD EN BASE A SU COMPLEJIDAD CICLOMÁTICA EN PROCEDIMIENTOS STREAM EN NODE.JS A TRAVÉS DE PRUEBA DE CAJA NEGRA

Para Optar al Título de Licenciatura en Ingeniería de Sistemas

MENCIÓN: GESTIÓN Y PRODUCCIÓN

Postulante: Orlando Moisés Flores Huanca

Tutor Metodológico: Ing. Marisol Arguedas Balladares

Tutor Revisor: Ing. Elías Carlos Hidalgo Mamani

Tutor Especialista: Ing. Julio Cesar Valdez Arzabia

EL ALTO – BOLIVIA

2020

DEDICATORIA

A ti que gobiernas el mundo y el basto universo que los seres humanos apenas nos esforzamos en entender y conocer.

A ustedes Anastasio y Rosa que se esforzaron por compartirme su sabiduría.

A mis hermanos y hermanas, cuya compañía hace muy valiosa la estadía en esta vida.

AGRADECIMIENTO

Agradecer a todos los docentes que imparten conocimiento con pasión y vocación, en especial al profesor Zenón Cartagena que me motivó durante mi formación secundaria e impulsó con sus consejos y recomendaciones en el campo de la matemática.

Contenido

DEDICATORIA i
AGRADECIMIENTO ii
ÍNDICE DE FIGURAS viii
ÍNDICE DE GRÁFICOS ix
ÍNDICE DE TABLAS x
ÍNDICE DE CUADROS xi
CAPITULO I 1
1. MARCO PRELIMINAR 1
1.1 INTRODUCCIÓN 1
1.2 ANTECEDENTES 2
1.2.1 Antecedentes Internacionales 2
1.2.2 Antecedentes Nacionales 5
1.3 PLANTEAMIENTO DEL PROBLEMA 6
1.3.1 Problema Principal 8
1.3.2 Problemas secundarios 8
1.3.3 Formulación del problema 8
1.4 OBJETIVOS 8
1.4.1 Objetivo General 8
1.4.2 Objetivos específicos 9
1.5 HIPÓTESIS 9
1.5.1 Conceptualización de Variables 9
1.5.2 Operacionalización de Variables 9
1.5.3 Docimasia de hipótesis 10
1.6 JUSTIFICACIÓN 10

1.6.1 Justificación Científica 10
1.6.2 Justificación Técnica 10
1.6.3 Justificación Económica 11
1.6.4 Justificación Social 11
1.7 METODOLOGÍA 11
1.7.1 Método Científico 12
1.7.2 Método de Ingeniería 12
1.8 HERRAMIENTAS 13
1.8.1 Temporizador de recursos Síncronos, Asíncronos y Rendimiento 13
1.8.2 Motor de JavaScript Node.js 13
1.8.3 Pruebas de Caja Negra 13
1.9 LÍMITES Y ALCANCES 13
1.9.1 Límites 13
1.9.2 Alcances 14
1.10 APORTES 14
CAPITULO II 15
2. MARCO TEÓRICO 15
2.1 MEDICIÓN Y MÉTRICAS DE SOFTWARE 15
2.1.1 Definición de Medición de Software 15
2.1.2 Definición de Métrica de Software 16
2.1.3 Métricas de Proceso, Producto, Predicción 17
2.1.4 Complejidad de Diseño y Número ciclomático de McCabe 18
2.1.4.1 Definición de Complejidad Ciclomática de un programa 25
2.1.5 Medición y Métricas de Confiabilidad de Productos de Software 26
2.1.5.1 Medición de confiabilidad 26

2.1.5.2 Métricas de Confiabilidad 27
2.1.5.3 Disponibilidad 29
2.2 CONFIABILIDAD DE PRODUCTOS DE SOFTWARE 30
2.2.1 Definición de Confiabilidad de Productos de Software 30
2.2.2 Falla de Sistema 31
2.3 FLUJOS DE DATOS (STREAM) EN NODE.JS 35
2.3.1 Definición 36
2.3.2 Asincronismo en streams 38
2.3.2.1 Asincronismo en procesos E/S 39
2.3.3 Formas de uso común de Streams 40
2.3.3.1 Streams del protocolo HTTP 40
2.3.3.2 Streams para el sistema de archivos 49
2.4 PRUEBAS DE SOFTWARE 51
2.4.1 Prueba de Caja Negra 52
CAPITULO III 53
3. MARCO APLICATIVO 53
3.1 DESCRIPCIÓN DEL CONTEXTO EN CONTROL DEL DESARROLLO Y PRODUCCIÓN DE SOFTWARE 53
3.2 REQUERIMIENTOS FUNCIONALES Y NO FUNCIONALES 53
3.2.1 Requerimientos funcionales 53
3.2.1.1 Métricas de complejidad ciclomática 53
3.2.1.2 Métricas de rendimiento 53
3.2.1.3 Prueba de software de caja negra 54
3.2.1.4 Medición del grado confiabilidad 54
3.2.2 Requerimientos no funcionales 54

3.2.2.1 Adaptabilidad 54
3.3 DISEÑO CON METODOLOGÍA EXPERIMENTAL 54
3.3.1 Datos Iniciales 54
3.3.1.1 Planteamiento general del problema 54
3.3.1.2 Hipótesis 54
3.3.1.3 Objetivo 55
3.3.1.4 Zona Geográfica 55
3.3.1.5 Recursos de Investigación 55
3.3.1.6 Precisión 55
3.3.2 Muestreo 56
3.3.2.1 Selección de usuarios y códigos fuente para su medición 56
3.3.3 Instrumentos de Investigación 56
3.3.3.1 Marcadores de rendimiento de node.js. 56
3.3.3.2 Capturas de Excepciones en el Código Fuente 56
3.3.3.3 Prueba de caja negra 57
3.4 DESARROLLO DE LA TÉCNICA DE MEDICIÓN DE CONFIABILIDAD . 57
3.4.1 Procesamiento de datos 57
3.4.2 Método de Análisis 60
3.4.3 Análisis de Datos 61
3.4.3.1 Resultados de las demandas hechas al Sistema de Pruebas de Código Fuente 61
3.4.3.2 Descripción de los Resultados del Sistema de Pruebas 61
3.4.3.3 Resultados del Sistema de Pruebas sobre el Rendimiento 62
3.4.3.4 Tablas de Frecuencias en Excel Sobre Resultados del Rendimiento Registrado. 64

3.4.3.5 Prueba de Hipótesis 68
3.4.4 Informe Final 70
3.4.4.1 Resumen 70
3.4.4.2 Probabilidad de falla a Demanda PDFAD 70
3.4.4.3 Diferencia de Medias de Rendimiento 70
3.4.4.4 Discusión 71
CAPITULO IV 72
4. CALIDAD Y SEGURIDAD 72
4.1 OBTENCIÓN DE RESULTADOS DE CALIDAD 72
4.2 CICLO DE VIDA DEL MODELO DE PRUEBAS 72
4.2.1 Estimación de Pagos al Personal 73
4.2.2 Tiempo Estimado de Trabajo 73
CAPITULO V 74
5. COSTO BENEFICIO 74
5.1 COSTOS 74
5.2 BENEFICIO 75
CAPITULO VI 76
6. CONCLUSIONES Y RECOMENDACIONES 76
6.1 CONCLUSIONES 76
6.2 RECOMENDACIONES 76
BIBLIOGRAFÍA 78
MANUAL DEL USUARIO 81
MANUAL TÉCNICO 85

ÍNDICE DE FIGURAS

Figura No. 2.1 NOTACIÓN DE FLUJO DE GRAFOS PARA CONSTRUCTOS ESTRUCTURADOS 20

Figura No. 2.2 (a) DIAGRAMA DE FLUJO (b) FLUJO DE GRAFO 22

Figura No. 2.3 LÓGICA DE COMPOSICIÓN 23

Figura No. 2.4 PATRONES DE USO DE SOFTWARE 33

Figura No. 2.5 MAPEO DE UN SISTEMA DE ENTRADA Y SALIDA.................... 34

Figura No. 2.6 ESTRUCTURA DEL MOTOR JAVASCRIPT NODE 38

Figura No. 2.7 EJEMPLO DE FLUJO ASÍNCRONO 39

Figura No. 2.8 CAPTURA DE LA EJECUCIÓN DEL SERVIDOR DE ARCHIVO ESTÁTICO. 45

Figura No. 3.1 CAPTURA DEL REPORTE DE RESULTADOS SOBRE EL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO PAOLOROSSI............. 63

Figura No. 3.2 CAPTURA DEL REPORTE DE RESULTADOS SOBRE EL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO DASPINOLA................ 64

Figura No. 3.3 HERRAMIENTA PARA IMPORTAR DATOS EXTERNOS A LA PLANILLA DE CALCULO EXCEL... 66

ÍNDICE DE GRÁFICOS

Gráfico No. 3.1 EJEMPLO DE GRAFICO DEL TIPO LINEA, EN EL SERVIDOR LOCAL CON CHART.JS 60

Gráfico No. 3.2 CAPTURA DEL REPORTE DE RENDIMIENTO DE CÓDIGO FUENTE DEL USUARIO PAOLOROSSI 61

Gráfico No. 3.3 CAPTURA DEL REPORTE DE RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO DASPINOLA 62

Gráfico No. 3.4 VALORES CRITICO Y DE PRUEBA EN LA DISTRIBUCIÓN NORMAL.. 69

ÍNDICE DE TABLAS

Tabla No. 1.1 OPERACIONALIZACIÓN DE VARIABLES 9

Tabla No. 2.2 ESPECIFICACIÓN DE DISPONIBILIDAD..................................... 30

Tabla No. 3.3 RECURSOS DE INVESTIGACIÓN 55

Tabla No. 3.4 SELECCIÓN DE CÓDIGO FUENTE EN EL REPOSITORIO DE PROYECTOS GITHUB 56

Tabla No. 3.5 TAMAÑO Y FORMATO DE ARCHIVOS PARA ALIMENTAR LA CAJA NEGRA 57

Tabla No. 3.6 RESUMEN DE PRUEBAS DE CÓDIGO FUENTE 61

Tabla No. 3.7 DATOS DE LA TABLA DE FRECUENCIAS DE LA PRUEBA DEL CÓDIGO DEL USUARIO DASPINOLA 67

Tabla No. 3.8 TABLA DE FRECUENCIAS DEL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO: DASPINOLA 67

Tabla No. 3.9 DATOS DE LA TABLA DE FRECUENCIAS DE LA PRUEBA DEL CÓDIGO DEL USUARIO PAOLOROSSI 68

Tabla No. 3.10 TABLA DE FRECUENCIAS DEL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO PAOLOROSSI 68

Tabla No. 3.11 PROBABILIDAD DE FALLA A DEMANDA DE LOS SUJETOS DE PRUEBA 70

Tabla No. 4.12 EJEMPLO DE CALCULO DE PLANILLA DE PAGOS AL PERSONAL INVOLUCRADO EN EL PROYECTO 73

Tabla No. 5.13 ESTIMACIÓN DE COSTO BENEFICIO EN UN PERIODO DE 12 MESES 74

ÍNDICE DE CUADROS

Cuadro No. 2.1 CREACIÓN DE SERVIDOR BÁSICO "HOLA MUNDO" 41

Cuadro No. 2.2 CREACIÓN DE UN SERVIDOR BÁSICO "PONG"...................... 42

Cuadro No. 2.3 CONTENIDO DEL ARCHIVO INDEX.HTML............................... 43

Cuadro No. 2.4 CONTENIDO DEL ARCHIVO STYLE.CSS................................. 43

Cuadro No. 2.5 CONTENIDO DEL ARCHIVO SCRIPT.JS.................................. 44

Cuadro No. 2.6 CONTENIDO DEL ARCHIVO SERVER.JS 44

Cuadro No. 2.7 CONTENIDO DEL CÓDIGO DE UNA PETICIÓN HTTPS. 46

Cuadro No. 2.8 CÓDIGO DE PETICIÓN HTTPS A LA API DE LA NASA 47

Cuadro No. 2.9 VARIACIÓN DEL CÓDIGO FUENTE DE PETICIÓN A LA API DE LA NASA 48

Cuadro No. 2.10 CÓDIGO DE PETICIÓN HTTPS A UNA API REST.................. 49

Cuadro No. 2.11 LECTURA ASÍNCRONA 51

Cuadro No. 2.12 CÓDIGO DE LECTURA DE ARCHIVO SÍNCRONO 51

Cuadro No. 3.1 EJEMPLO DE CÓDIGO FUENTE CON APLICACIÓN DE MARCADORES DE RENDIMIENTO 58

Cuadro No. 3.2 EJEMPLO DE CONTENIDO DEL ARCHIVO DE REGISTROS EN NOTACIÓN DE OBJETOS JAVASCRIPT UN ARCHIVO CON DIFERENTES REGISTROS 59

Cuadro No. 3.3 PROGRAMA AUXILIAR PARA EXPORTACIÓN DE DATOS DE MARCAS DE TIEMPO A FORMATO TEXTO, SEPARADO POR COMAS 65

CAPITULO I

MARCO PRELIMINAR

CAPITULO I

1. MARCO PRELIMINAR

1.1 INTRODUCCIÓN

Los procesos que corresponden a desarrollar y producir software, son parte esencial de las labores cotidianas en el rubro de la ingeniería de software, administrar y controlar los procesos de desarrollo y producción de software también debería ser algo cotidiano, pero esto muchas veces no siempre resulta real, la importancia de administrar y controlar procedimientos de software radica en la posibilidad coadyuvar en el aseguramiento de la calidad de los productos de software.

Gracias a los avances del desarrollo del control de calidad de software que se basan en el desarrollo de control de calidad en la industria de productos físicos en general, donde la evaluación de calidad se realiza tomando en cuenta el efecto del medio ambiente sobre el hardware como ser corrosión, humedad, tiempo de vida útil de componentes físicos, etc. Mientras que en desarrollo producción de software no hay efectos del medio ambiente sobre el normal y continuo funcionamiento, sin embargo, esa normalidad de funcionamiento es establecido principalmente por la complejidad de diseño de procedimientos desarrollados que necesariamente deben ser administrados y controlados.

Debido a que muchas veces existe incertidumbre acerca de la confiabilidad de procedimientos de software y la necesidad de administración y control de procesos de desarrollo, se puede a través de la técnica que se propone en ese trabajo controlar el proceso de desarrollo o producción de software usando para ello métricas de funcionalidad, rendimiento y complejidad ciclomática sobre algún procedimiento y derivando medidas que serán evaluadas estadísticamente para contar con registros acerca del procedimiento en cuestión.

Para el uso de la técnica de medición y métricas, en este trabajo se usará Node, que terminó siendo JavaScript del lado Servidor, que incluye módulos nativos que

sirven para la implementación de registros de marcas de tiempo sobre procesos con un reloj monotónico de alta resolución con las especificaciones de la W3C[1].

1.2 ANTECEDENTES

1.2.1 Antecedentes Internacionales

- En el año 2011, fue presentado en la Facultad de Graduación y Estudios Post Doctorales (Ingeniería Eléctrica y de Computadoras) de la Universidad Británica Columbia (Vancouver), la tesis para la obtención del grado de “Master of Applied Science” con el título “Characterizing and refactoring asynchronous JavaScript callbacks”. Gallaba (2015).

  En este trabajo se alude a las aplicaciones web modernas y el uso extensivo de JavaScript, del cual se estima que es uno los lenguajes ampliamente usado en el mundo. El uso de callback´s[2] como una característica popular en JavaScript. Sin embargo, son una fuente problemas de comprensión y mantenibilidad. En el mencionado trabajo se estudia muchas características del uso de callback´s a través de un gran número aplicaciones de JavaScript y se encontró en ese estudio, que más del 43% de todas las funciones que llaman sitios, que aceptan callback´s son anónimas, la mayoría de los callback´s están anidados, y más de la mitad de todos los callback´s son invocados asíncronamente.

  También hace mención a las Promesas (una variación de la estructura de devolución de llamadas), como una alternativa para la compleja composición del flujo ejecución asíncrona y como un mecanismo robusto para la revisión de errores en JavaScript. Se observa el uso de callbacks para construir una herramienta que rediseñe funciones callbacks en Promesas. Se resalta la técnica y herramienta es aplicable ampliamente a un rango variado de aplicaciones JavaScript.

---

[1] https://w3c.github.io/perf-timing-primer/ especificaciones de reloj monotónico de alta resolución.

[2] Es una función pasada como un argumento dentro de otra función ejecutada al finalizar su ejecución.

- También por el año 2017, se presenta en la "Mid Sweden University", carrera Ingeniería de la Computación, la Tesis de licenciatura, Proyecto de grado independiente con el título "Parallelism in Node.js applications, Data flow analysis of concurrent scripts". Jansson (2017).

  Hace mención al total uso de procesadores multinúcleo en aplicaciones Node.js, las aplicaciones deben ser programas como procesos múltiples. La ejecución paralela puede incrementar el rendimiento de datos por lo tanto menor almacenamiento en buffer de datos para comunicación de procesos internos. El modelo de programación asíncrono e interface para la operación del sistema hace conveniente las herramientas que son muy bien ajustadas para programación multiproceso. Sin embargo, el comportamiento en tiempo de ejecución de procesos asíncronos resulta en una carga y flujo de datos de un procesador no deterministas. Eso significa que la ganancia de rendimiento del incremento de concurrencia depende de ambos el tiempo de ejecución de la aplicación y la capacidad del hardware para ejecución paralela.

  Esta tesis tiene como objetivo explorar los efectos de incrementar el paralelismo de las aplicaciones en Node.js por las diferencias de medición en las cantidades de datos del buffer cuando procesos distribuidos se ejecuten de un número variado de núcleos con un rango fijo de arribo de datos asíncronamente. La meta es simular y examinar el comportamiento del tiempo de ejecución de tres arquitecturas de aplicaciones multiproceso en Node.js en atención a debatir y evaluar técnicas de paralelismo de software. Las tres arquitecturas son: nodos entubados para procesamiento dependiente temporalmente, un vector de nodos para procesamiento de datos paralelos, y una rejilla de nodos para procesamiento ramificado uno a varios.

  Para visualizar el comportamiento en tiempo de ejecución en ese trabajo se propone, un entorno simulado de procesos usando múltiples procesos de Node.Js, la simulación es basada en agentes, donde el agente es una abstracción para un flujo de datos específicos dentro de una aplicación

distribuida, donde los procesos se comunican asíncronamente a través de mensajes vía sockets.

Los resultados de ese trabajo, muestran que el rendimiento puede incrementar, se distribuyen múltiples aplicaciones Node.js a través de múltiples procesos corriendo en paralelo en hardware multinúcleo[3]. Habrá sin embargo un retorno disminuido como el número de procesos activos igual o se exceda el número de núcleos. Una buena regla parece distribuir la lógica desacoplada a través de muchos procesos como haya núcleos. La interacción entre procesos asíncronos es en todo hecho muy simple con Node.js. Aunque corriendo múltiples instancias de Node.Js requiere mucha memoria, la arquitectura distribuida tiene el potencial de incrementar rendimiento por la cercanía como las muchas veces como el número de núcleos en el procesador.

- Otro trabajo el año 2016, es presentado en la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid, el trabajo de fin de grado con el título "Creación y visualización de métricas ágiles mediante el uso de herramientas mashup[4]". Rodríguez Fraga (2016).

  El trabajo tiene como base de estudio a los equipos de desarrollo de software que trabajan siguiendo metodologías ágiles y ellos utilicen diversas herramientas online para sus tareas de gestión de proyecto, control de código o integración continua. Plantea el ejemplo de herramientas como son Jira, GitHub o Jenkins. El problema del escenario es que el equipo genera una gran cantidad de información de proyecto totalmente dispersa, y resulta por tanto indispensable su posterior recopilación para poder monitorizar el proceso de desarrollo y generar las métricas ágiles necesarias para su análisis. Este trabajo propone una solución a este problema usando la herramienta de mashup Wirecloud, desarrollando una serie de componentes

[3] Se refiere a dispositivos que combinan dos o más microprocesadores independientes, a menudo en un solo circuito integrado.

[4] En desarrollo web, es una forma de integración y reutilización. Ocurren cuando una aplicación web es usada o llamada desde otra aplicación.

que permiten a los usuarios configurar de forma flexible la información que obtienen y cómo la representan, permitiendo generar distintas métricas ágiles que se ajusten a sus necesidades. Los componentes desarrollados se dividen en distintas categorías:

- Harvesters, que se ocupan de obtener la información de las herramientas ágiles online.
- Splitters, cuya función es obtener propiedades concretas de los datos obtenidos por los harvesters.
- Componentes que realizan transformaciones sobre listas, encargados de agrupar, filtrar los datos y realizar operaciones sobre los datos.
- Componentes de Representación gráfica, cuya función es transformar los datos de forma que estos representen gráficas o tablas de datos y mostrarlos al usuario.

Los componentes de Wirecloud envían los datos que generan entre sí mediante unas conexiones configurables, de forma que permite (al ser los componentes desarrollados genéricos, aceptando una gran variedad de datos de entrada) generar salidas que se ajusten a las necesidades concretas del proyecto analizado sin tener que desarrollar nuevos componentes.

1.2.2 Antecedentes Nacionales

- En el contexto departamental nacional, el año 2006, es que se presenta en la carrera de Informática de la Facultad de Ciencias Puras y Naturales en la Universidad Mayor de San Andrés, la tesis de grado, con el título “Modelo de Ingeniería del Mantenimiento para Productos Software Orientado a Objetos” caso “Proyectos de grado de la Carrera de Informática U.M.S.A.”. Cuevas Callisaya (2006).

  El trabajo se centra en la descripción de la ingeniería de software como un enfoque sistemático disciplinado y cuantificable para la construcción, operación y mantenimiento del software y las fases del ciclo de vida del producto de software que son análisis, diseño, codificación, pruebas y

mantenimiento. El trabajo se enmarca en la última fase, se muestra un conjunto de tareas a realizar, antes, durante y después del proceso de mantenimiento denominado modelo de ingeniería del mantenimiento, basado en la teoría de la metodología del mantenimiento Mantema y en el estándar 12207, propone un prototipo desarrollado para la evaluación de la calidad del software se encuentra bajo los lineamientos del paradigma orientado a objetos, utilizando el método de proceso unificado racional (RUP) para su construcción y como herramienta de diseño el lenguaje unificado (UML).

- Otra investigación, el año 2011, se presenta en la carrera de Informática de la Facultad de Ciencias Puras y Naturales en la Universidad Mayor de San Andrés, la tesis de grado, con el título "Elaboración de métricas para evaluar la seguridad del software durante su desarrollo". Huanca Aliaga (2011).

  El documento, considera a la seguridad del software como la seguridad de la información comúnmente conocida, la cual incluye temas como: la protección de los sistemas de información en contra del acceso no autorizado y la modificación de información. El objetivo del ciclo de vida de desarrollo de un producto software y las métricas asociadas al mismo, es desarrollar un producto que posea la calidad necesaria y suficiente para que satisfaga las necesidades y requerimientos del cliente. En ese trabajo se presenta un marco conceptual de calidad y se propone un modelo, métodos, procedimientos, criterios y herramientas a emplear para medir la seguridad de un producto software y posteriormente realizar las correspondientes conclusiones y recomendaciones que nos sirvan para la retroalimentación y mejora del producto software. Además, se establecerán los medios necesarios a emplear en el proceso de definición y medición de las características y atributos principales que van a formar el modelo de calidad.

1.3 PLANTEAMIENTO DEL PROBLEMA

La confiabilidad de una porción de código, está relacionada a la complejidad del diseño con el cual fue concebido, fijando límites y alcances de la funcionalidad de

ese código. En la fase de concepción se amolda la porción de código con algún grado de tolerancia al éxito o fracaso que depende del establecimiento e implementación de la interpretación de especificaciones y requerimientos transformados en forma de código fuente por algún programador.

Producir y desconocer el grado de confiabilidad de software y la influencia de los factores que ejercen su aumento o disminución como lo es la complejidad ciclomática, podría tener leves o serias consecuencias en relación al tipo de sistema y el contexto donde será usado. Por ejemplo, un sistema en un hospital o un sistema de navegación aéreo requieren un alto grado de confiabilidad en cuanto a demandas y transacciones del sistema y no es así en un sistema de procesador de imágenes o texto donde es más importante asegurar el mayor tiempo de funcionamiento sin falla ni pérdida de datos.

Entregar un producto de software desconociendo el grado de confiabilidad, tiene además el efecto de la aceptación o rechazo de parte de los consumidores, lo cual involucra la credibilidad y seriedad de la entidad en la que se origina el producto.

El eje central de esta investigación gira en base al uso de una técnica y la evaluación de confiabilidad de procedimientos stream[5] sobre su complejidad ciclomática en Node.Js. Para ello en este estudio, se aplican métricas de confiabilidad y complejidad ciclomática sobre procedimientos stream, para evaluar la relación e influencia con procesos estadísticos. La variable independiente complejidad ciclomática de procedimiento de stream, puede definirse como complejidad de lógica del flujo de control de procedimientos stream en Node.js; la variable dependiente es probabilidad de éxito o fracaso sobre el rendimiento de un proceso o producto en un periodo de tiempo en condiciones específicas.

Conocer el grado de confiabilidad de un producto de software antes de su entrega es información útil y necesaria para tomar decisiones adecuadas para la categorización de productos y sectores de consumidores en contextos apropiados.

[5] Flujo de datos continuo

1.3.1 Problema Principal

El núcleo de un lenguaje de programación es el lugar donde se desarrollan y desenvuelven procedimientos con algún grado de complejidad sobre el diseño de esa porción de código fuente.

Los procesos que se involucran en resolver el problema principal que representa la incertidumbre de la confianza de procedimientos stream, son las métricas de complejidad y funcionalidad en cuanto al tiempo de resolución de procesos y la medición del grado de complejidad del procedimiento.

La causa del problema viene dada por la falta de aplicación de técnicas de administración y control del desarrollo y producción de software en procedimientos ya desarrollados o producidos.

1.3.2 Problemas secundarios

- No se cuenta con un registro del comportamiento funcional de procedimientos stream.
- Existe incertidumbre de la probabilidad de confianza de procedimientos stream.
- No se asegura la confiabilidad de procedimientos stream basado en su complejidad ciclomática.

1.3.3 Formulación del problema

¿Existe diferencia entre los grados de confiabilidad de procedimientos stream en base a su complejidad ciclomática que permita coadyuvar a la toma de decisiones adecuada, para asegurar la calidad del producto de software en Node.js?

1.4 OBJETIVOS

1.4.1 Objetivo General

Aplicar técnicas de medición y métricas para comparar la igualdad o diferencia entre grados de confiabilidad de procedimientos stream en base a su complejidad ciclomática para coadyuvar al aseguramiento de la calidad del producto de software en Node.js para la toma de decisiones correspondientes.

1.4.2 Objetivos específicos

- Usar métricas de rendimiento con pruebas de caja negra para probar y registrar la funcionalidad de un procedimiento stream.
- Aplicar técnicas estadísticas para medir, comparar la diferencia de los grados confiabilidad de procedimientos stream.
- Asegurar la confiabilidad de un procedimiento stream basado en su complejidad ciclomática.

1.5 HIPÓTESIS

Existirá diferencia significativa entre los grados de confiabilidad de procedimientos stream, basado en su complejidad ciclomática para coadyuvar a la toma de decisiones adecuada para asegurar la calidad del producto de software en Node.js.

1.5.1 Conceptualización de Variables

Variable Independiente. - Complejidad lógica del flujo de control de código fuente sobre streams en Node.js.

Variable Dependiente. - Probabilidad de éxito sobre el rendimiento de un proceso o producto en un periodo de tiempo en condiciones específicas.

1.5.2 Operacionalización de Variables

Tabla No. 1.1 OPERACIONALIZACIÓN DE VARIABLES

Variable	Definición Conceptual	Dimensión	Indicadores	Índices
Complejidad ciclomática de procedimientos (variable independiente)	Complejidad lógica del flujo de control de código fuente sobre streams en Node.js	Número de complejidad ciclomática del procedimiento	Simple	Bajo (1-4)
			Bien estructurado y estable	Bajo (5-10)
			Más complejo	Moderado (11-20)
			Complejo preocupante	Alto (21-50)
Confiabilidad del procedimiento (variable dependiente)	Probabilidad de éxito sobre el rendimiento de un proceso o producto en un periodo de tiempo en condiciones específicas	Tiempo medio para fallar TMPF, en sistemas de transacción larga en relación al tiempo medio de uso del sistema TMUS por el usuario sin grabar su trabajo	No es confiable	TMPF<TMUS
			Es confiable	TMPF>TMUS
		Probabilidad de falla a demanda PDFAD, en	Excesivamente riesgoso	1/10 falla/demandas

		sistemas críticos cuando una falla a demanda puede guiar a serios fallos del sistema	Muy riesgoso	1/100 fallas/demandas
			Riesgoso	1/1000 fallas/demandas
			Poco riesgoso	1/10000 fallas/demandas
			Sin riesgo significativo	1/100000 fallas/demandas
		Tasa de ocurrencia de fallos TOCF, en sistemas con demanda regular en vez de demanda intermitente	Alta	100 fallas/tiempo
			Media	10 fallas/tiempo
			Baja	1 falla/tiempo
			Alta	100/1000 fallas/transacción
			Media	10/1000 fallas/transacción
			Baja	1/1000 fallas/transacción

Fuente: Elaboración propia basado en Sommerville (2016)

1.5.3 Docimasia de hipótesis

Existirá diferencia significativa entre los grados de confiabilidad de procedimientos stream, basado en su complejidad ciclomática para coadyuvar a la toma de decisiones adecuada para asegurar la calidad del producto de software en Node.js.

H0: No existe diferencia significativa entre grados de confiabilidad de procedimientos stream, significativa (<95%) en base a su complejidad ciclomática.

H1: Existe diferencia significativa entre grados de confiabilidad de procedimientos stream (>95%) en base a su complejidad ciclomática.

1.6 JUSTIFICACIÓN

1.6.1 Justificación Científica

El desarrollo de planes y programas de medición y métrica de productos de software, no es acompañado con la evolución del hardware. Existen muchos intentos por uniformar las mediciones y métricas existentes sin llegar a acuerdos generales. Los consensos logrados, fueron realizados en forma regional por lo cual es necesaria la promoción de la investigación y el desarrollo sobre medición y métricas de software para la evaluación en producción de software de calidad.

1.6.2 Justificación Técnica

Evaluar la confiabilidad de procedimientos en ejecución que son realizados en forma instantánea y en tiempo real, requiere la implementación de instrumentación

necesaria para ello a manera de sensores que capturan el registro de tiempo, que en esta investigación representan las métricas de rendimiento. Controlar el comportamiento de productos de software en tiempo de ejecución, requiere un esfuerzo adicional por la instrumentación de precisión requerida.

1.6.3 Justificación Económica

Aunque implementar planes de medición y métricas sobre productos de software, requiere aumento de costos adicionales en el proceso de desarrollo. Liberar y entregar productos con un estándar de confiabilidad conocido otorga la credibilidad y aceptación que tiene adjunto la confianza del segmento de usuarios del producto de software, que implica asegurar la aceptación de actualizaciones y nuevos productos que tienen un retorno económico asegurado.

1.6.4 Justificación Social

Conocer el grado de confiabilidad de productos de software puede guiar al desarrollo en atención a demandas de los clientes en algunos casos requerirán sistemas con una alta demanda o transacciones regulares, en otros casos es importante cubrir la demanda de sistemas con mayor seguridad en el tiempo de ejecución y la pérdida de datos.

En general es necesario poner atención en la producción de software atendiendo sus características de calidad y categorización, ya que van destinados a una gran masa de usuarios que requieren de productos de software diseñados con responsabilidad social y satisfacción del cliente.

1.7 METODOLOGÍA

La metodología usada para el presente trabajo, se basa sobre la investigación experimental, que se entiende como un proceso lógico, metódico y ordenado de procedimientos secuenciales para realizar una investigación científica, que consiste en la manipulación rigurosamente controlada de variables experimentales, no comprobadas o condicionadas, con las que se pretenden analizar y describir el comportamiento de un fenómeno o problema en su campo de acción. El objetivo es identificar las causas que producen una determinada conducta, una situación específica o un acontecimiento particular.

1.7.1 Método Científico

La base científica del presente documento, está asentada sobre un modelo general que Muñoz Razo (2011), propone para optimizar el desarrollo de la investigación. Este modelo comprende, en el presente caso de tesis, desde la elección del tema hasta la presentación del borrador inicial de la misma. Otros elementos sólo se mencionan para complemento del propio modelo.

- Fase I Datos iniciales.
- Fase II Muestreo.
- Fase III Instrumentos de la investigación.
- Fase IV Procesamiento de datos.
- Fase V Métodos de análisis.
- Fase VI Análisis de datos.
- Fase VII Informe final

1.7.2 Método de Ingeniería

El método de ingeniería usado para este trabajo de investigación corresponde a la aplicación adecuada de métricas de producto y medición de confiabilidad realizadas a través de instrumentación con módulos nativos de Node.js para registrar tiempo de precisión, dicho proceso dentro de la ingeniería de software, también es conocido como pruebas de software que se realizan previo al lanzamiento de un producto de software, para la detección de errores y su corrección, todo esto en el presente trabajo es usado para la obtención de datos dentro del proceso general de investigación.

Otra herramienta es la aplicación de pruebas de caja negra donde se da atención a la funcionalidad sin poner atención a la estructura interna del código, detalles de implementación o ejecución interna del software, en este tipo de prueba se pone atención a las entradas y salidas del sistema.

1.8 HERRAMIENTAS

1.8.1 Temporizador de recursos Síncronos, Asíncronos y Rendimiento

La plataforma Node.js implementa módulos para realizar el rastreo del tiempo y comportamiento síncrono y asíncrono de la ejecución del código fuente con un reloj monotónico de alta resolución con las especificaciones de la W3C.

1.8.2 Motor de JavaScript Node.js

Node.Js llega a ser JavaScript del lado servidor, que es el lenguaje que se usará en esta investigación debido a su capacidad de alto tráfico de datos y sobretodo en su diseño que permite la manipulación de procedimientos stream y una alta concurrencia de eventos. Además de contar con un ecosistema de recursos muy amplio.

1.8.3 Pruebas de Caja Negra

Las pruebas de caja negra permiten conocer aspectos relevantes a la funcionalidad de entradas y salidas del sistema sin tomar en cuenta el diseño del código fuente.

1.9 LÍMITES Y ALCANCES

1.9.1 Límites

Esta investigación está limitada a aplicar métricas sobre algunos atributos internos y externos de software como ser la complejidad del diseño y el código, rendimiento y además de la medición de confiabilidad de procedimientos stream, entre otras características de calidad sobre los atributos externos del software existentes como ser: seguridad, resiliencia, robustez, comprensibilidad, adaptabilidad, modularidad, complejidad, portabilidad, usabilidad, reusabilidad, eficiencia, capacidad de aprendizaje. En cuanto a los atributos internos existen otros atributos como ser: cohesión, capacidad de acoplamiento, funcionalidad del código, métricas orientadas a clases y objetos, que no son contemplados en el presente trabajo.

También existe la limitación sobre aquellas métricas relacionadas al proceso del proyecto que también está relacionada con el desarrollo de proyectos de ingeniería de software.

1.9.2 Alcances

Uno los alcances de esta investigación es abordar con una aproximación a la medición y métricas de confiabilidad y complejidad ciclomática con un nivel de confiabilidad significativo, que tienen a su vez como objetivo identificar procedimientos de software confiables y rechazar aquellos procedimientos de software que no sean confiables, todo eso es inherente a la complejidad en cualquier sistema o módulo basado en el problema, que necesita ser resuelto antes de la entrega del producto de software, lo que garantiza la calidad del mismo producto. Sea cualquiera el problema, se quiere minimizar la complejidad de la solución.

Por otro lado, también se ofrece una aproximación hacia la confiabilidad general de un sistema, junto a la confiabilidad del hardware y la confiabilidad del operador del sistema, pero enfocándonos en más propiamente en el producto de software y las variaciones que podría sufrir.

1.10 APORTES

Puede considerarse un aporte, la implementación de una técnica para la evaluación de confiabilidad de procedimientos en Node.js. Por otro lado, como incentivo o referencia a aficionados, programadores, ingenieros de software y quienes están involucrados en la producción de software, hacia la implementación de planes y programas de medición y métricas de software para la evaluación y revisión de calidad de productos de software, siendo la aplicación de estas técnicas de suma importancia en el desarrollo y la producción, asimismo en el proceso apropiarse de la responsabilidad social para producir software confiable.

CAPITULO II

MARCO TEÓRICO

CAPITULO II

2. MARCO TEÓRICO

2.1 MEDICIÓN Y MÉTRICAS DE SOFTWARE

Para poder comprender acerca de los conceptos que comúnmente son usados en ingeniería de software y especialmente en el rubro las métricas y de las pruebas de software, es necesario desglosar lo que algunos autores definen acerca del tema.

2.1.1 Definición de Medición de Software

El autor Nicolette (2015), define medición de software como: Una medición es una observación cuantitativa de los siguientes aspectos: algo relevante a las decisiones por hacer, de Información que se tiene que reportar relacionado al progreso del desarrollo, o de los efectos de mejoramiento de un proceso (p. 2).

Con ese mismo propósito Fenton y Bieman (2014), llegan a la siguiente definición: es el proceso por el que números o símbolos son asignados a atributos de entidades en el mundo real de tal manera que son descritas de acuerdo a reglas claramente definidas (p. 5). Y en forma complementaria indican que es una reducción de la incertidumbre expresada cuantitativamente basada en una o más observaciones (p. 8).

El autor Pressman (2013), llega a la siguiente definición: Es un elemento clave de procesos de ingeniería, se pueden usar para entender mejor los atributos de los modelos que se crean y para valorar la calidad de los productos o sistemas sometidos a ingeniería que se construyen, pero a diferencia de otras disciplinas de ingeniería (las ciencias puras), la del software no está asentada en las leyes cuantitativas de la física, puesto que las mediciones y métricas del software con frecuencia son indirectas, están abiertas a debate (p. 526).

En su más reciente libro el autor Sommerville (2016), define la medición de software como: una cuantificación de algún atributo de sistema de software (P. 717), tal como su complejidad o confiabilidad. Por la comparación de los valores medidos y los estándares que se aplican en una organización, obtener conclusiones sobre la

calidad de software o evaluación de la efectividad de herramientas y métodos y procesos de software. En un mundo ideal, la administración de la calidad podría basarse en la medición de los atributos que afectan a la calidad de software. Usted podría entonces evaluar objetivamente procesos y cambiar herramientas que permitan mejorar la calidad del software.

Entonces se puede llegar a la siguiente definición sobre medición de software: es la observación cuantitativa sobre la calidad de sistemas de software; es justificable decir que es una observación cuantitativa debido a que proporciona un valor que puede ser comparado. E involucra la calidad de sistemas de software, sistemas como la conjunción de partes del modelo.

Aunque los términos medición y métrica son usados de ida y vuelta, hay una inclinación de los autores citados anteriormente a usar el término medición a grados superiores sobre los modelos.

2.1.2 Definición de Métrica de Software

El autor Nicolette (2015), define métrica como: Una métrica es una medición recurrente que tiene algún tipo de poder informacional, diagnóstico, motivacional, o predictivo**.** Ayuda a comprender si estas en riesgo de perder resultados esperados o si las prácticas o cambios en progreso están resultando en la mejora del rendimiento (p. 2).

Por otro lado, Fenton y Bieman (2014), definen métrica como un término y actividades relacionadas: Las métricas de software es un término que abarca muchas actividades, las cuales involucran algún grado de medición de software (p. 17).

Continuando con las percepciones de la definición de nuestros autores, se tiene a Pressman y Maxim (2015), quienes citando al IEEE definen métrica como: El glosario estándar del IEEE de terminología de ingeniería de software define métrica como una medida cuantitativa del grado al cual un sistema, componente, o proceso posee un atributo dado (p. 655).

Sommerville (2016), define métrica como característica de un sistema de software, documentación de sistema, o proceso de desarrollo que puede ser objetivamente medido (p. 718).

Haciendo una síntesis de los autores citados, es posible elaborar la siguiente definición de métrica de software: Medición cuantitativa y recurrente del grado de un atributo dado; se justifica que es un proceso de medición cuantitativa y recurrente por el uso de la métrica misma como un instrumento de medición sobre algún atributo (código fuente, diseño, cohesión, acoplamiento, confiabilidad, usabilidad, reusabilidad, etc.) y brinda la posibilidad de realizar una comparación con otras mediciones de resultados anteriores, de ese mismo proceso sobre algún atributo del producto de software.

2.1.3 Métricas de Proceso, Producto, Predicción

En los textos de ingeniería de software, frecuentemente se encuentran los términos: "métricas de proceso", "métricas de producto", "métricas de predicción", para referirse al uso de las métricas en ciertos aspectos del desarrollo de software.

Las métricas de proceso y proyecto de software son medidas cuantitativas que permiten obtener comprensión acerca de la eficacia del proceso del software y de los proyectos que se realizan, usando el proceso como marco conceptual**.** Se recopilan datos básicos de calidad y productividad**.** Luego, se analizan, se comparan con promedios anteriores y se valoran para determinar si han ocurrido mejoras en calidad y productividad. Las métricas también se usan para puntualizar áreas problemáticas, de modo que puedan desarrollarse remedios y el proceso de software pueda mejorarse (Pressman, 2013, p. 571).

Métricas de proceso, usado por desarrolladores para deducir información sobre el proceso del software. Una métrica típica es el tipo de eficiencia o faltas de detección durante el desarrollo que es el rango del número de fallas detectadas durante el desarrollo al total de numero de fallas detectadas en el producto en su vida útil (Schach, 2011, p. 133).

Las métricas de control o proceso, apoyan el proceso de administración (Sommerville, 2016, p. 718).

El término métricas de control o proceso al parecer es usado para referirse a una evaluación del proceso general del proyecto asociado también a la productividad del personal involucrado.

Las métricas de producto, son métricas para medir atributos internos de un sistema software (tamaño del sistema, la medida en líneas de código, número de métodos asociados. Se dividen en dos: Métricas dinámicas y métricas estáticas (Sommerville y Campos Olguin, 2011, p. 672).

Las métricas de producto son métricas de predicción usadas para cuantificar atributos internos de un sistema de software (Sommerville, 2016, p. 721).

Métricas de producto, miden aspectos del producto en sí mismo, tal como tamaño, confiabilidad. (Schach, 2011, p. 133)

Las métricas de predicción, se asocian con el software en sí y a veces se conocen como métricas de producto. Ejemplo: Complejidad ciclomática, longitud promedio de identificadores de un programa, número de atributos y operaciones asociados con las clases de objetos en un diseño (Sommerville y Campos Olguin, 2011, p. 668).

Las métricas de predicción ayudan a predecir características del software. (Sommerville, 2016, p. 718)

El término "métrica de producto" parece ser más apropiado para referirse a una evaluación de los atributos de internos y externos del producto de software y el ´termino "métricas predicción", parece más apropiado para referirse a las pruebas con cargas de datos y demandas realizadas al producto de software.

2.1.4 Complejidad de Diseño y Número ciclomático de McCabe

La prueba de McCabe puede ser calculado casi tan fácil como las líneas de código LOC. En algunos casos ha mostrado ser una buena métrica para predecir defectos. Por ejemplo, Schach (2011), citando a otro autor Walsh (1979) analizó 276 módulos

en el sistema Aegis[6]. Midiendo la complejidad ciclomática, M, el encontró que 23 por ciento de los módulos con M mayor o igual a 10 tienen el 21 por ciento más defectos por línea de código que los módulos con valor M pequeños. Sin embargo, la validez de las métricas de McCabe han sido cuestionadas seriamente por los fundamentos teóricos y en las bases de muchos experimentos diferentes citados por Schach (2011).

Schach (2011) citando a otros autores que analizaron los datos disponibles en densidad de defectos. Concluyeron que muchas métricas de complejidad, incluyendo la de McCabe, muestra una alta correlación con el número de líneas de código, o más precisamente, el número de instrucciones entregables, ejecutables. En otras palabras, cuando los investigadores miden lo que ellos creen ser la complejidad de un artefacto de código o un producto, el resultado que obtienen puede ser ampliamente un reflejo del número de líneas de código, una medida que se asocia fuertemente con el número de defectos. En suma, las métricas de complejidad proveen pequeñas mejoras a las líneas de código para predecir tasas de defectos. (Schach, 2011, p. 528)

La más famosa métrica de complejidad y podría decirse la más popular, es la complejidad ciclomática de McCabe, la cual es una medida del número de flujos de control dentro de un módulo. La teoría subyacente es que, a mayor número de caminos en un módulo, mayor complejidad. La métrica de McCabe fue originalmente diseñada para indicar la capacidad de prueba y comprensibilidad. Está basada en una teoría clásica de grafos, en la que se calcula el número ciclomático de un grafo, denotado por V(g), por contar el número rutas linealmente independientes dentro de un programa, que permite determinar el número mínimo de pruebas únicas que pueden ser ejecutadas a cada declaración ejecutable. Un módulo es definido como un conjunto de código ejecutable que tiene una entrada y una salida. El número ciclomático puede ser calculado en dos maneras (que dan el

[6] https://www.lockheedmartin.com/en-us/products/aegis-combat-system.html un sistema de combate naval.

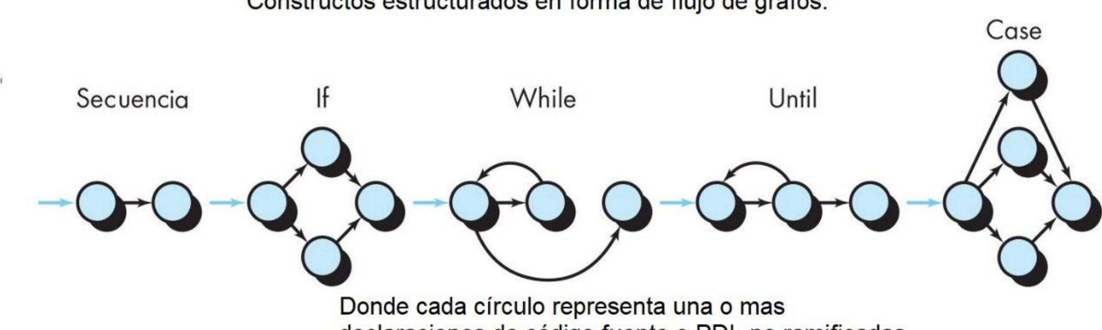
mismo resultado), si por el conteo de nodos y lados del grafo o por el conteo de puntos de decisión binaria. Esto es:

V(g) = e – n + 2, donde g es el grafo de control del módulo, e es el número de lados, y n es el número de nodos.

V(g) = bd + 1, donde bd es el número de decisiones binarias en el grafo de control. Si hay n-maneras de decisión, es contado como n-1 decisiones binarias.

Se espera que módulos con alta complejidad ciclomática, sean más difíciles de probar y mantener, debido a su alta complejidad, y módulos con baja complejidad ciclomática, sean fáciles. (Laird y Brennan, 2006, p. 58)

Figura No. 2.1 NOTACIÓN DE FLUJO DE GRAFOS PARA CONSTRUCTOS ESTRUCTURADOS



Fuente: (Pressman y Maxim, 2015, p. 502)

Una contribución clave de McCabe, ha sido su transposición del número ciclomático de la teoría de grafos al software. En el software, el programa se modela como un gráfico de flujo de control, que es una estructura abstracta utilizada en los compiladores; específicamente una representación abstracta de un procedimiento o programa, mantenida internamente por un compilador.

- Cada vértice del gráfico representa un bloque básico.
- Los bordes dirigidos se usan para representar saltos en el flujo de control.

Hay dos bloques especialmente designados:

- El bloque de entrada, a través del cual el flujo de control entra en el gráfico de flujo, y
- El bloque de salida, a través del cual sale todo el flujo de control.

Las gráficas de flujo de control de programas no están fuertemente conectadas, pero lo están cuando se añade un borde virtual, conectando el nodo de salida con el nodo de entrada Abran (2010) citando a Watson (1995).

Dado que el gráfico de flujo de control en el caso del software se transforma entonces en un gráfico fuertemente conectado, el número ciclomático del gráfico puede aplicarse a esta representación de programas.

Así, el número ciclomático, cuando se aplica al software en esta transposición, se convierte:

$$v(G) = e - n + p + 1\, borde\ virtual$$

Nota: se ha añadido uno a la anterior, para integrar el borde virtual suplementario.

Además, en esta transposición, sólo se tienen en cuenta los módulos individuales, en lugar de toda la entidad de software. En la situación particular definida por McCabe, el número de componentes conectados p es siempre igual a 1. McCabe define entonces la ecuación anterior como:

$$v(G) = e - n + 2$$

(Abran, 2010, pp. 134–135)

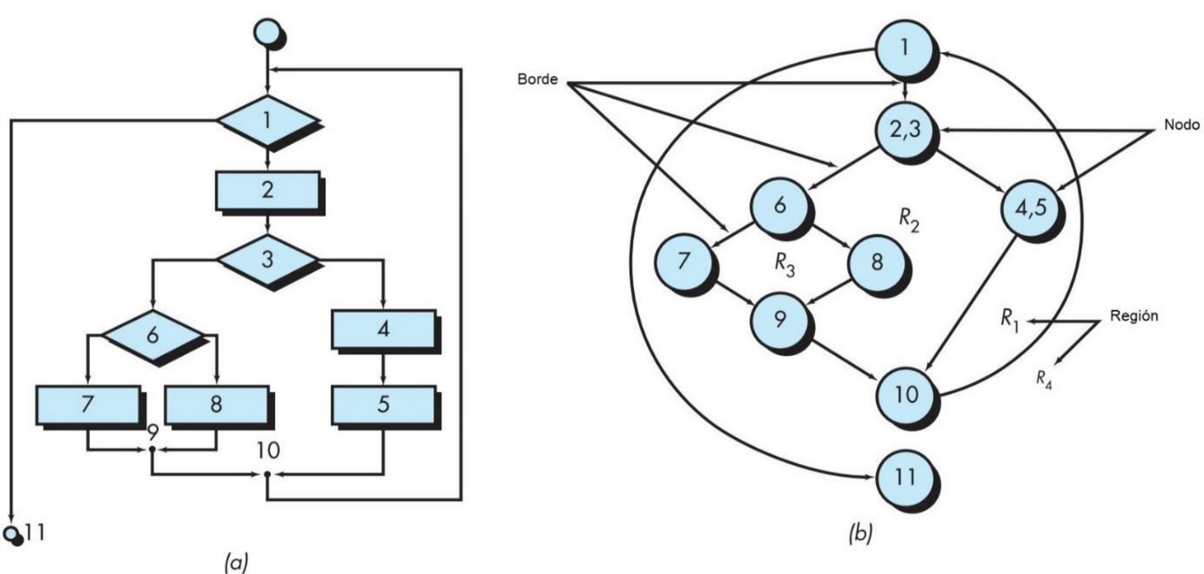
Para una mejor comprensión la ecuación:

$$v(G)^{ciclo\ independiente} = e^{borde} - n^{nodo} + p^{componentes\ conectados} + 1^{borde\ virtual}$$

quedaría de la siguiente forma:

$$v(G)^{ciclo\ independiente} = (e+1)^{borde-borde\ virtual} - n^{nodos} + p^{componentes\ conectados}$$

Figura No. 2.2 (a) DIAGRAMA DE FLUJO (b) FLUJO DE GRAFO



Fuente: (Pressman y Maxim, 2015, p. 502)

La complejidad ciclomática tiene un fundamento en la teoría de grafos y le proporciona una métrica de software extremadamente útil. La complejidad se calcula de una de tres maneras:

1. El número de regiones del gráfico de flujo corresponde a la complejidad ciclomática.
2. La complejidad ciclomática V (G) para un gráfico de flujo G se define como

$$V\,(\,G\,)\;=\;E\;-\;N\;+\;2$$

   donde E es el número de bordes de la gráfica de flujo y N es el número de nodos de la gráfica de flujo.
3. La complejidad ciclomática V (G) para un gráfico de flujo G también se define como

$$V\,(\,G\,)\;=\;P\;+\;1$$

   donde P es el número de nodos predicados contenidos en el gráfico de flujo G.

Remitiéndose una vez más al gráfico de flujo de la Figura No. 2.2, la complejidad ciclomática puede calcularse utilizando cada uno de los algoritmos que se acaban de señalar:

1. El gráfico de flujo tiene cuatro regiones.
2. V (G) = 11 bordes - 9 nodos + 2 = 4.
3. V (G) = 3 nodos predicados + 1 = 4.

(Pressman y Maxim, 2015, p. 504)

Figura No. 2.3 LÓGICA DE COMPOSICIÓN

Fuente: (Pressman y Maxim, 2015, p. 503)

Cuando se encuentran condiciones compuestas en un diseño de procedimiento, la generación de un gráfico de flujo se hace ligeramente más complicado. Una condición compuesta se produce cuando uno o más operadores booleanos (lógico O, Y, NAND, NOR) están presentes en una declaración condicional. Refiriéndonos a la Figura No. 2.3, el segmento del lenguaje de diseño de programas (PDL) se traduce en el gráfico de flujo que se muestra. Observe que se crea un nodo separado para cada una de las condiciones a y b en la sentencia IF a OR b. Cada nodo que contiene una condición se denomina nodo predicado y se caracteriza por tener dos o más aristas que emanan de él (Pressman y Maxim, 2015, p. 503).

El test de ruta base[7] es una técnica de prueba de caja blanca propuesta por primera vez por Tom McCabe. El método de la trayectoria base permite al diseñador del caso de prueba derivar una medida de complejidad lógica de un diseño de procedimiento y utilizar esta medida como guía para definir un conjunto básico de trayectorias de ejecución. Los casos de prueba derivados para ejercitar el conjunto base están garantizados para ejecutar cada declaración en el programa al menos una vez durante la prueba (Pressman y Maxim, 2015, p. 501).

Una ruta independiente es cualquier ruta a través del programa que introduce al menos un nuevo conjunto de declaraciones de procesamiento o una nueva condición. Cuando se establece en términos de un gráfico de flujo, un camino independiente debe moverse a lo largo de al menos un borde que no ha sido atravesado antes de que el camino sea definido (Pressman y Maxim, 2015, p. 503).

Por ejemplo, un conjunto de caminos independientes para el gráfico de flujo ilustrado en la Figura No. 2.2 es:

Ruta **1**: 1-11

Ruta **2**: 1-2-3-4-5-10-1-11

Ruta **3**: 1-2-3-6-8-8-9-10-1-11

Ruta **4**: 1-2-3-6-7-9-10-1-11

Tengan en cuenta que cada nuevo camino introduce un nuevo borde. El camino

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

no se considera un camino independiente porque es simplemente una combinación de caminos ya especificados y no atraviesa ningún borde nuevo.

Las rutas 1 a 4 constituyen un conjunto de bases para el gráfico de flujo de la Figura No. 2.2. Es decir, si se pueden diseñar pruebas para forzar la ejecución de rutas (un conjunto de bases), se habrá garantizado la ejecución de cada declaración del programa al menos una vez y cada condición se habrá ejecutado en sus lados

[7] Es una forma de uso, con el que se conoce a la complejidad ciclomática.

verdadero y falso. Cabe señalar que el conjunto de bases no es único. De hecho, se pueden derivar varios conjuntos de bases diferentes para un diseño de procedimiento determinado (Pressman y Maxim, 2015, pp. 503–504).

La importancia en realizar el análisis de rutas, se debe a la posibilidad de conocer las áreas lógicas que cubre el código fuente de cada procedimiento y la posibilidad de diseñar pruebas para cada área lógica y forzar el funcionamiento de esas áreas.

2.1.4.1 Definición de Complejidad Ciclomática de un programa

La complejidad ciclomática propuesta por McCabe cuenta con bastante aceptación como una métrica de la complejidad del diseño de un programa; es necesaria entender como la definen los autores, todos enfocados al software, para definir complejidad ciclomática Laird y Brennan (2006, p. 58) indican que es una medida del número de flujos de control dentro de un módulo.

Por otro lado Schach (2011, p. 491) indica que es el número de decisiones binarias (predicados) mas 1 o equivalentemente el número de ramas en el artefacto del código.

En ese enfoque de definición, Sommerville (2016, p. 722) define que es una medida de la complejidad de control de un programa.

Asimismo, Mili y Tchier (2015, p. 315) indica que es un reflejo de la complejidad de sus estructura de control.

En ese sentido, Abran (2010, p. 134) asevera que es la complejidad de control de flujo de grafo.

Con un enfoque sobre el número ciclomático Fenton y Bieman (2014, p. 73) indican que es un mapeo de los diagramas en números reales.

En un sentido estructural se puede decir que la complejidad ciclomática de un programa es la complejidad de estructuras de control de un programa.

2.1.5 Medición y Métricas de Confiabilidad de Productos de Software

La producción de software tiene una clara relación entre las métricas dinámicas[8] y las características de calidad de software. Es bastante fácil medir el tiempo de ejecución requerido para una función particular y evaluar el tiempo requerido para iniciar un sistema. Esas funciones se relacionan directamente con la eficiencia del sistema. Similarmente el número de fallas del sistema y el tipo de falla puede ser registrado y relacionado directamente a la confiabilidad del software.

Una de las características de calidad de software es la confiabilidad que en combinación con métricas dinámicas nos otorgan mediciones del tiempo de éxito o fracaso ocurrido en la ejecución de un sistema de software.

2.1.5.1 Medición de confiabilidad

Para medir la confiabilidad de un sistema, se tiene que recolectar datos sobre sus operaciones, los datos requeridos pueden incluir según Sommerville (2016):

1. El número de fallas de sistema dados un número de solicitudes para servicios de sistema. Es usado para medir la PDFAD[9] y aplicar independientemente del tiempo durante el que se realizan las demandas.

2. El tiempo o el número de transacciones entre fallas de sistemas más el total de tiempo transcurrido o el número total de transacciones. Este es usado para medir el TDOCDF[10] y TMPF[11].

3. El tiempo de reparación o reinicio TMPR[12] después de la falla de sistema que induce a la pérdida del servicio. Este es usado para la medición de disponibilidad. La disponibilidad no solo depende del tiempo entre fallas sino también del tiempo requerido para traer de vuelta al sistema a su operación.

---

[8] Son métricas que se realizan en forma activa en tiempo de ejecución del software

[9] Probabilidad de Falla a Demanda

[10] Tasa de Ocurrencia de Fallos

[11] Tiempo Medio Para Fallos

[12] Tiempo Medio Para Reparación

2.1.5.2 Métricas de Confiabilidad

Sommerville (2016), hace una aproximación a las métricas de confiabilidad usando tiempo o demandas como también hace referencia a las solicitudes hechas al sistema.

A) Probabilidad de Falla a Demanda PDFAD

Es una métrica con la que se puede definir la probabilidad que una demanda para servicio desde un sistema resultará en una falla de sistema. Así que si, PDFAD = 0.001 significa que hay un chance 1/1000 que una falla ocurrirá cuando una demanda es hecha Sommerville (2016, p. 314).

Es una métrica que tiene utilidad para hacer encontrar fallas en pruebas de sistemas críticos que podrían derivar en daños físicos o desastres mayores.

Debe ser usado en situaciones cuando una falla a demanda puede guiar a serios fallos de sistema**.** Esto aplica independientemente de las demandas. Por ejemplo, un sistema de protección que monitorea un reactor químico y apaga su reacción si es sobrecalentado debe tener esta confiabilidad especificada usando PDFAD. Generalmente, las demandas en un sistema de protección son poco frecuentes tanto que el sistema es la última línea de defensa, después de todo todas las estrategias de recuperación han fallado. Por lo tanto, un PDFAD de 0.001 (1 falla en 1000 demandas) podría parecer riesgoso. Sin embargo, si hay solo dos o tres demandas en el sistema en todo su ciclo de vida, entonces el sistema es poco probable que falle alguna vez (Sommerville, 2016, pp. 314–315).

B) Tasa de Ocurrencia de Fallos TDOCDF

Esta métrica establece el probable número de fallos del sistema que son probablemente observados relacionados a un cierto periodo de tiempo (una hora), o al número de ejecuciones del sistema. Del ejemplo PDFAD, se extrae que la TDOCDF 1/1000. El recíproco de la TDOCDF es el tiempo medio para falla (TMPF), que es a veces usado como una métrica de fiabilidad. TMPF, es el número promedio de unidades de tiempo observados entre fallos de sistema. A una TDOCDF de dos

fallas por hora implica que el tiempo medio para falla es 30 minutos (Sommerville, 2016, p. 314).

La métrica TDOCDF debe ser usada cuando las demandas en los sistemas son hechas regularmente en vez de intermitentemente. Por ejemplo, en un sistema que maneja un gran número de transacciones, puede especificar una TDOCDF de 10 fallos por día. Esto significa que usted estará dispuesto a aceptar que unos promedios de 10 transacciones por día no se completarán exitosamente y tendrán que ser cancelados y reenviados nuevamente. Alternativamente puede especificar TDOCDF como el número de fallas por 1000 transacciones. (Sommerville, 2016, p. 315)

C) Tiempo Medio Para Fallos TMPF

Si el tiempo absoluto entre fallas es importante, puede especificar la confiabilidad como el tiempo medio para fallos (TMPF). Por ejemplo, si está especificando la confiabilidad requerida para un sistema con transacciones largas (tal como un sistema de diseño guiado por computador), usted debe usar esta métrica. El TMPF debe ser más grande que el tiempo promedio que un usuario trabaja en su modelo sin guardar los resultados de usuario. Esto significa que no es probable que los usuarios pierdan su trabajo a través de la falla del sistema en cualquier sesión. (Sommerville, 2016, p. 315)

Se puede apreciar que las métricas presentadas anteriormente, también tienen la utilidad de definir las pruebas para el tipo de sistema donde pueden especificarse la demanda como es PDFAD, las transacciones TDOCDF y el tiempo de uso con TMPF e incluirse en los requerimientos para su programación y codificación en la fase temprana de la concepción de un producto de software o sistema.

D) Tiempo Medio Entre Fallos TMEF

Un sistema o producto de software funciona con éxito durante un tiempo, y luego falla. Las medidas que se han introducido hasta ahora se han centrado en la interrupción del uso exitoso. Sin embargo, una vez que se produce un fallo, se pierde más tiempo al localizar y reparar los fallos que lo provocan. Por lo tanto, es

importante saber el tiempo medio de reparación (TMDR) de un componente que ha fallado. La combinación de este tiempo con el tiempo medio hasta la falla nos dice cuánto tiempo el sistema no está disponible para su uso: el tiempo medio entre fallas (TMEF) es según Fenton y Bieman (2014) simplemente:

TMEF = TMPF + TMDR

Sin embargo, el TMEF puede ser problemático por dos razones: 1) proyecta un lapso de tiempo entre fallos, pero no nos proporciona una tasa de fallos proyectada, y 2) el TMEF puede interpretarse erróneamente como el promedio de vida útil, aunque esto no es lo que implica (Pressman y Maxim, 2015, p. 461).

E) Tiempo Medio Para Reparación TMPR

Cuando un producto falla, un problema importante es el tiempo que tarda, en promedio, repararlo (tiempo medio de reparación). Pero, a menudo más importante es el tiempo que se tarda en reparar los resultados del fallo. Este último punto se pasa por alto frecuentemente (Schach, 2011, p. 164).

2.1.5.3 Disponibilidad

Disponibilidad AVAIL es la probabilidad de que un sistema esté en funcionamiento cuando se hace una demanda de servicio. Por lo tanto, una disponibilidad de 0,9999 significa que, en promedio, el sistema estará disponible durante el 99,99% del tiempo de funcionamiento. En la Tabla No. 2.2, se muestra lo que significan en la práctica los diferentes niveles de disponibilidad.

Tabla No. 2.2 ESPECIFICACIÓN DE DISPONIBILIDAD

Disponibilidad	Explicación
0.9	El sistema estará disponible 90% del tiempo, lo que significa que en un periodo de tiempo de 24 horas (1440 minutos) el sistema estará fuera de servicio por 144 minutos.
0.99	En un periodo de 24 horas el sistema estará fuera de servicio por 14,4 minutos
0.999	El sistema estará fuera de servicio por 84 segundos en un periodo de 24 horas
0.9999	El sistema no está disponible durante 8,4 segundos en un período de 24 horas, más o menos, un minuto por semana

Fuente: (Sommerville, 2016, p. 314)

Además de una medida de confiabilidad, también debería desarrollar una medida de disponibilidad. La disponibilidad de un programa es la probabilidad de que un programa esté operando de acuerdo a los requerimientos en un momento dado y se define como:

Disponibilidad = (TMPF/(TMPF+TMPR)) *100%

La medida de fiabilidad TMEF es igualmente sensible al TMPF y al TMPR. La medida de disponibilidad es algo más sensible al TMPR, una medida indirecta de la mantenibilidad del software (Pressman y Maxim, 2015, p. 461).

2.2 CONFIABILIDAD DE PRODUCTOS DE SOFTWARE

La confiabilidad de los productos de software tiene bases en la teoría general de la confiabilidad que fue desarrollada a la par de los procesos que evolucionaron junto con la invención y producción de productos físicos en la historia humana.

2.2.1 Definición de Confiabilidad de Productos de Software

Habiendo estudiado un poco acerca lo que es medición y métricas de software para tener aproximaciones, en la descripción de comportamientos de los productos de software, se puede continuar con la exploración de la confiabilidad de productos de software, poniendo atención en las definiciones que algunos autores nos dan.

Según Mili y Tchier (2015) se refiere a confiabilidad de un producto de software como un reflejo, en forma general de la probabilidad de que el producto funcione sin fallos durante largos períodos de tiempo (p. 289).

Por otro lado, el autor, Abran (2010), indica que según la ISO 9126, confiabilidad es la capacidad del producto de software para mantener un nivel de rendimiento específico cuando se utiliza en condiciones específicas (p. 217).

Así también, Sommerville (2016), se refiere a la confiabilidad de un sistema como la probabilidad que sobre un periodo de tiempo dado, el sistema entregará correctamente servicios como se espera por el usuario (p. 289).

Con ideas parecidas, Schach (2011) explica que confiabilidad es una medida de la frecuencia y gravedad de fallar de un producto (p. 164).

De igual forma Fenton y Bieman (2014), lo definen en términos de rendimiento operacional, algo que claramente no se puede medir antes que el producto esté terminado (p. 47).

Tomando en cuenta las definiciones de cada autor, se puede efectuar una interpretación de las definiciones de confiabilidad como probabilidad de éxito sobre el rendimiento de un proceso o producto de software, en un periodo de tiempo en condiciones específicas. Tal medida es producto de la aplicación de métricas de confiabilidad sobre algún atributo de producto de software que influye las probabilidades de éxito ya sea en transacciones o intervalos de tiempo de ejecución de demandas o transacciones realizadas y la probabilidad de fallas que podría presentarse en el futuro con las demandas y transacciones del sistema, para esto es muy importante la frecuencia de ocurrencia de fallos, el tiempo promedio de recuperación del sistema. Considerando esas probabilidades de éxito y fracaso en cada proceso o sistema, también es importante el estudio de aquello a lo que se llama falla.

2.2.2 Falla de Sistema

Las fallas de sistema podrían ocurrir de forma aleatoria en el futuro, pero la probabilidad de que un sistema falle dependerá de la interpretación e

implementación fiel de la concepción de requerimientos de cada proceso o sistema, en cada producto de software para el entorno en el cual desempeñará la función para la cual fue concebido. En otro entorno quizás no se podrá observar el mismo desempeño debido a la tolerancia de los requerimientos de cada sistema.

La confiabilidad de un sistema, no es un valor absoluto, depende de dónde y cómo el sistema es usado. Por ejemplo, se puede decir que se mide la confiabilidad de una aplicación en un ambiente de oficina, donde muchos usuarios no están interesados en la operación del software. Ellos siguen instrucciones para su uso y no intentan experimentar con el sistema. Si entonces usted mide la confiabilidad del mismo sistema en un ambiente de universidad, entonces la confiabilidad puede ser bastante diferente. Aquí, los estudiantes pueden explicar las bondades del sistema y usarlo de maneras inesperadas. Esto puede resultar en fallas de sistema que no ocurren en el más limitado ambiente de oficina. Sin embargo, las percepciones de la confiabilidad del sistema en cada uno de esos ambientes son diferentes (Sommerville, 2016, p. 311).

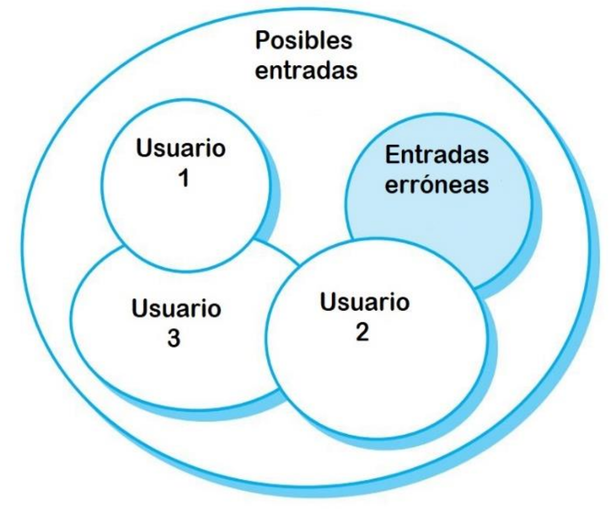
La definición de confiabilidad está basada en la idea positiva de una de operación libre de fallo, cuando las fallas son eventos externos que afectan a usuarios de un sistema y son producidas por los mismos usuarios. Pero que constituye "falla". Una definición técnica de falla es comportamiento que no está conforme a las especificaciones del sistema (Sommerville, 2016, p. 311). Entonces de acuerdo a esa definición, se tiene a las especificaciones como el elemento que limita y posibilita la introducción de fallos en un sistema.

Una realidad que hay que asumir en el proceso de producción de software, es que nadie excepto los desarrolladores de software, leen el documento de las especificaciones de software. Los usuarios, por lo tanto, anticipan que el software debe comportarse de una manera cuando la especificación dice algo completamente diferente. Es así que, las especificaciones de software son con frecuencia incompletas o incorrectas, y se les deja a los ingenieros de software interpretar como el sistema debe comportarse. Como ellos no son expertos en el dominio, ellos pueden no implementar el comportamiento que los usuarios esperan.

El software puede comportarse como es especificado, pero, para los usuarios, sigue fallando (Sommerville, 2016, p. 311). Tal realidad es muy frecuente y se puede percibir lastimosamente cuando ya se hizo la entrega del producto.

La falla, por lo tanto, no es algo que pueda ser objetivamente definido. Más bien, es un juicio hecho por los usuarios de un sistema. Esta es una razón por la que los usuarios, no todos tienen la misma impresión de la confiabilidad de un sistema.

Figura No. 2.4 PATRONES DE USO DE SOFTWARE



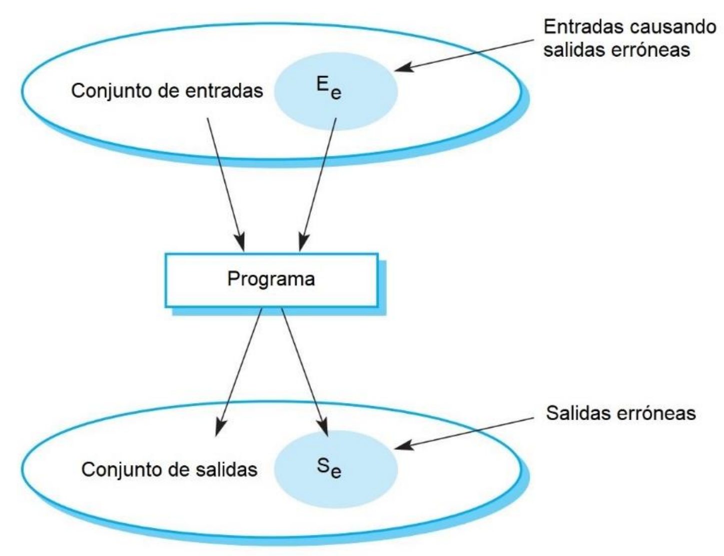
Fuente: (Sommerville, 2016, p. 312)

Los fallos que afectan a la fiabilidad del sistema para un usuario pueden no aparecer nunca en el modo de trabajo de otro. En la Figura No. 2.4, el conjunto de entradas erróneas corresponde a la elipse etiquetada “Ee” de la Figura No. 2.5. El conjunto de entradas producidas por el Usuario 2 se cruza con este conjunto de entradas erróneas. Por lo tanto, el Usuario 2 experimentará algunas fallas en el sistema. Sin embargo, el Usuario 1 y el Usuario 3 nunca utilizan las entradas del conjunto erróneo. Para ellos, el software siempre parecerá fiable. (Sommerville, 2016, p. 312)

Muchas entradas no guían a la falla del sistema, sin embargo, algunas entradas o combinaciones de entradas, ver Figura No. 2.5, causan fallas del sistema o salidas erróneas para ser generadas. La confiabilidad del programa depende del número de entradas de sistema del que son miembros del conjunto de entradas que guían a una salida errónea, en otras palabras, el conjunto de entradas que causan que el

código defectuoso sea ejecutado y errores de sistema ocurran. Si las entradas del conjunto erróneo son ejecutadas por el código que raramente es usado, entonces los usuarios casi nunca verán fallos (Sommerville, 2016, p. 312).

Figura No. 2.5 MAPEO DE UN SISTEMA DE ENTRADA Y SALIDA



Fuente: (Sommerville, 2016, p. 311)

En otra línea de la confiabilidad y los posibles fallos que pudieran introducirse en un sistema, también es importante considerar que la disponibilidad de un sistema no solo depende del número de fallas de sistema, sino también del tiempo necesario para reparar los defectos que ha causado la falla. Por lo tanto, si el sistema A falla una vez al año y el sistema B falla una vez al mes, entonces A es aparentemente más confiable que B. Sin embargo, asumiendo que el sistema toma 6 horas para reiniciar después de una falla, mientras que el sistema B toma 5 minutos para reiniciar. La disponibilidad del sistema B en el año (60 minutos fuera de servicio) es mucho mejor que el sistema A (360 minutos fuera de servicio) (Sommerville, 2016, p. 312).

Además, la posible perturbación a los usuarios del sistema, que podría ser causada por los sistemas no disponibles no se refleja en la simple métrica de la disponibilidad que especifica el porcentaje de tiempo que el sistema está disponible. El horario en que el sistema falla también es importante. Si un sistema no está disponible durante una hora cada día entre las 3 y las 4 de la madrugada, esto puede no afectar a

muchos usuarios. Sin embargo, si el mismo sistema no está disponible durante 10 minutos durante la jornada laboral, la indisponibilidad del sistema tiene un efecto mucho mayor en los usuarios (Sommerville, 2016, p. 312).

Como es posible apreciar, la confiabilidad de un producto de software, no es solamente la medición de la probabilidad de éxito, también es importante la observación del origen de la introducción de las fallas a un sistema, que se podría decir que está en las fases tempranas del desarrollo de un proyecto de software como son las especificaciones y requerimientos, además de la concepción e interpretación de cada desarrollador sobre esas especificaciones y requerimientos, por esa razón es acertado indicar que la falla de un sistema es un juicio de los usuarios de un sistema, lo que sí es cuantificable es la frecuencia de fallos y la tendencia por la preferencia a usar el sistema en un entorno específico.

2.3 FLUJOS DE DATOS (STREAM) EN NODE.JS

Cuando Dahl diseña Node, JavaScript, no era el lenguaje original de su elección, todavía después de explorarlo el encontró un muy buen lenguaje moderno sin dar ninguna opinión sobre flujos de datos stream, sistema de archivos, manejo de objetos binarios, procesos, sistema de redes y otras capacidades que uno espera que existan en el sistema de un lenguaje de programación. JavaScript, estaba limitado al navegador, y no encontraba uso y no había implementado esas características.

Dahl, se fue guiado por unos pocos principios de diseño:

- Un programa/proceso de Node se ejecuta en un solo hilo, ordenando su ejecución a través de un ciclo de eventos.
- Las aplicaciones web son de intensas operaciones del tipo entrada y salida E/S, así que el enfoque debería ser en hacer operaciones E/S rápidas.
- El flujo de programa es siempre dirigido a través de callbacks asíncronos.
- Las operaciones costosas de CPU deben ser divididas en procesos paralelos separados emitiendo eventos como resultado de su arribo.
- Programas complejos deben ser ensamblados de programas más simples.

El principio general es, que las operaciones nunca deben ser bloqueadas. El anhelo de velocidad de Node de alta concurrencia y eficiencia o consumo mínimo de recursos tiene una demanda sobre la reducción de desperdicio. Un proceso de espera es un proceso de desperdicio, especialmente cuando se espera un proceso de E/S.

El diseño dirigido a eventos y la asincronía de JavaScript completa perfectamente este modelo. Las aplicaciones expresan interés en un evento futuro y son notificadas cuando ese evento ocurre.

El tiempo que toma una acción E/S en completarse es desconocido, así que el patrón es consultar una notificación cuando un evento E/S es emitido, cuando quiera que sea, permitiendo a otras operaciones ser completadas en ese tiempo.

2.3.1 Definición

Los streams provienen de los primeros días de Unix, y probaron ser, a través de décadas una manera confiable de componer grandes sistemas de pequeños componentes que lo hacen efectivamente. En Unix los streams son implementados en línea de comandos por el uso de "|" tuberías, en Node, el modulo empaquetado stream es usado por la librería central y puede ser usado por módulos espacio usuario. Similar a Unix, el operador primario de composición stream de Node es llamado .pipe() con eso se puede obtener un mecanismo de contrapresión libre para reducir el flujo de escritura para consumidores lentos[13].

El autor Antonio Laguna, hace una definición más simple, introduciendo el término pipe, que, aunque es una palabra muy corta en el idioma inglés, tiene mucha significación y es esencial comprender para entender lo que son los streams en Node el indica que son básicamente pipes[14].(Laguna, 2013, p. 60).

El autor David Mark Clements, en su libro a manera de recetario, hace la siguiente definición introduciendo algunos términos como son objeto y trozos que amplían la

13 Dispositivos con bajas capacidades de procesamiento, memoria y almacenamiento.

14 Tuberías o cadenas de proceso

idea principal de los streams de Node e índice que *es* básicamente un objeto con algunos métodos formalizados y funcionalidad, que está orientado a recibir, enviar y procesar datos en pequeñas piezas llamados trozos (Clements, 2014, p. 127).

De la base de conocimientos de la página principal de Node.js se extrae que Un stream es una abstracción del tipo entrada, salida E/S de Node (Node.js, 2011a).

También de la misma base de conocimiento de la página antes mencionada, se extrae una definición complementaria, agregando el concepto de codificación asíncrona e indica que son un constructo básico que fomenta la codificación asíncrona. Permiten procesar datos en cuanto es generado o recuperado. Pueden ser de lectura, escritura o ambos*.* (Node.js, 2011b)

La autora, Liz Parody, en su blog expande la definición de stream hacia un plano más general indicando que son métodos de manejo lectura/escritura de archivos, comunicaciones en red, o todo tipo de intercambio de información de extremo a extremo de una manera eficiente (Parody, 2019).

De todas estas definiciones se puede inferir nuestra propia definición como una interfaz abstracta del tipo entrada salida, que fomenta la codificación asíncrona y tiene métodos de manejo de lectura/escritura de archivos, comunicaciones en red, o todo tipo de intercambio de información de extremo a extremo de una manera eficiente.

Para justificar la definición propia asumida en este trabajo de investigación, se acepta que es una interfaz abstracta del tipo entrada salida, porqué de ella pueden derivar otras hechas a medida de las necesidades del desarrollador, como es una abstracción no es tan fácil de comprender y requiere de conocimientos del lenguaje c, c++ que es de donde se origina la mayoría de las interfaces.

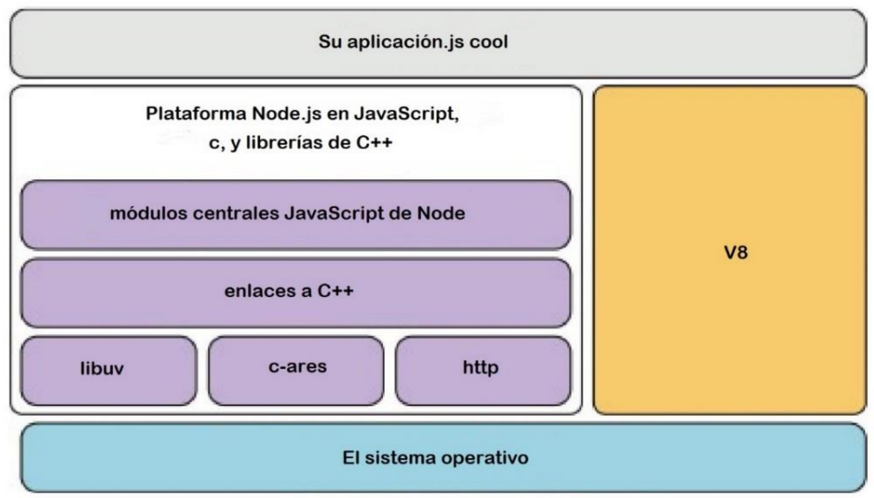
Es correcto afirmar que, fomenta la codificación asíncrona porqué la abstracción de la interfaz contiene métodos y eventos que permiten el uso, reúso y rediseño sobre código para flujo de datos, desde alguna parte y hacia otra parte, no bloquee de ninguna manera, alguna operación en el proceso de ejecución del código.

2.3.2 Asincronismo en streams

El asincronismo en JavaScript, es delegar tareas o funciones que toman más tiempo ejecutarlas, mientras otros procesos se están ejecutando, y como JavaScript fue diseñado para ejecutarse en el entorno web y en vista que podía ser capaz de soportar alta concurrencia de eventos, es que fue tomado como base para crear el motor JavaScript V8. De ahí que Node es diseñado para uso en entorno servidor y a veces cliente, usando las características de los sistemas operativos y sus beneficios.

La característica notable de V8 es compilar directamente a código máquina, e incluye características de optimización de código, que ayudan a Node a mantenerse rápido. La parte que administra E/S es libuv[15]; V8 administra la interpretación y ejecución de su código JS. Para usar libuv con V8, se usa una capa de enlace C++.

Figura No. 2.6 ESTRUCTURA DEL MOTOR JAVASCRIPT NODE



Fuente: (Young et al., 2017)

La manera en que las instrucciones escritas en un programa JavaScript son compiladas por el motor V8 en una lista de instrucciones cuyo contexto de ejecución es accesible a través el objeto nativo de Node process.

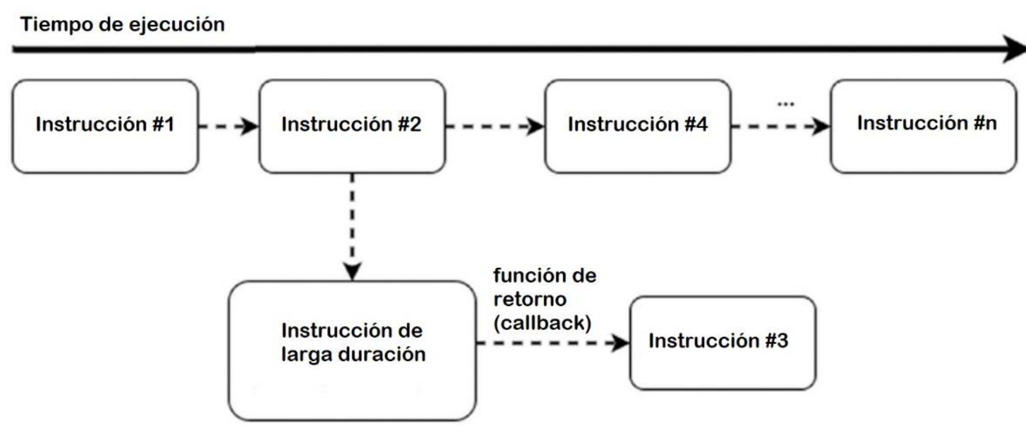
---

[15] Es una librería C++, de código abierto que en Node.js da acceso a la administración interna de sistemas operativos, redes y concurrencia.

El único hilo que forma la espina del bucle de eventos de Node es el bucle de eventos de V8. Cuando las operaciones E/S son iniciadas dentro de este ciclo son delegadas a libuv, que administra la solicitud usando su propio ambiente (asíncrono, multihilo). libuv anuncia la conclusión de las operaciones entrada, salida E/S, permitiendo a cualquier callback en espera en este evento a ser re-introducido al hilo principal V8 para su ejecución (Pasquali, 2013, p. 19).

Entonces se puede asumir que el asincronismo ver Figura No. 2.7 o la delegación de tareas es esencial en la estructura de Node como proceso fundamental y abstracto en las operaciones de comunicación y transmisión de datos E/S.

Figura No. 2.7 EJEMPLO DE FLUJO ASÍNCRONO



Fuente: (Doglio, 2016, p. 76)

2.3.2.1 Asincronismo en procesos E/S

Uno de los factores en los que Node.js tiene tantos adeptos y contribuciones en todo el mundo, es la forma del tratamiento de los flujos de datos stream de forma asíncrona que fue iniciada con JavaScript por el lado frontend[16], mejorada por Google con su versión V8, y establecida para el lado backend[17] con Node, usando la versión V8 de JavaScript.

[16] Desarrollo web del lado cliente o en el navegador.

[17] Desarrollo web del lado servidor.

2.3.3 Formas de uso común de Streams

Aunque el principio de diseño de las interfaces de Node están realizadas a bajo nivel para generalizar las aplicaciones y dar apertura a la imaginación de cada desarrollador, principalmente el uso generalizado de Node es como servidor, lo cual implica muchas funcionalidades que incluyen el procesamiento de protocolos de comunicaciones en línea HTTP, HTTPS y HTTP2, para lo que Node incluye diferentes módulos para cada uno.

Todas las funcionalidades en cuanto al manejo de streams continúan en cada uno de los módulos nativos que pueden ser invocados fácilmente en Node.

2.3.3.1 Streams del protocolo HTTP

La interfaz de administración del protocolo HTTP, es una interfaz de muy bajo nivel, para respaldar una amplia posibilidad de características cliente y servidor.

Es el módulo principal responsable por el servidor Node.js HTTP. Los principales métodos son los siguientes:

- http.createServer(): Retorna un nuevo objeto web servidor
- http.listen(): Comienza a aceptar conexiones en el puerto y host especificado
- http.createClient(): Crea un cliente y hace solicitudes a otros servidores
- http.ServerRequest(): Pasa solicitudes entrantes a manejadores de solicitudes
- data: Emitido cuando una parte del cuerpo del mensaje es recibido
- end: Emitido exactamente cada vez por cada solicitud
- request.method(): Retorna el método solicitud como una cadena
- request.url(): Regresa la cadena de solicitud URL
- http.ServerResponse(): Crea el objeto this internamente por un servidor HTTP —no por el usuario —y es usado como una salida para el manejador de solicitud
- response.writeHead(): Envía una cabecera de respuesta a la solicitud
- response.write(): Envía un cuerpo de respuesta
- response.end(): Envía y termina un cuerpo de respuesta

(Mardan, 2018, pp. 31–32)

Para poder apoyar todo el espectro de posibles aplicaciones HTTP, la API HTTP de Node.js es de muy bajo nivel. Se ocupa sólo del manejo de los streams y del análisis sintáctico de los mensajes. Analiza un mensaje en los encabezados y el cuerpo, pero no analiza los encabezados o el cuerpo. ("HTTP | Node.js v14.5.0 Documentation," 2020b)

A) Stream HTTP Servidor

HTTP es un protocolo de transferencia de datos sin estado construido sobre un modelo de petición/respuesta. Los clientes hacen peticiones (request) a los servidores, que luego devuelven una respuesta (response). Facilitar este tipo de comunicación de red de patrones rápidos es el tipo de Node de entrada salida E/S, diseñado, como es lógico, como un conjunto de herramientas para crear servidores.

Para ejemplificar la funcionalidad como servidor que tiene Node.js, se usan los siguientes:

- El típico hola mundo

Cuadro No. 2.1 CREACIÓN DE SERVIDOR BÁSICO "HOLA MUNDO"

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.write('Hola Mundo...');
    res.end();
});
server.listen(4000);
```

Fuente: (Teixeira, 2012, p. 55)

En la primera línea se trae el módulo 'http' en el que se llama a createServer, para crear una instancia del servidor HTTP.

Entonces se realiza la escucha a los tipos de eventos 'request', pasando una función de respuesta (callback), con dos argumentos: el objeto request y el objeto response.

Entonces se puede usar el objeto response para escribir al cliente (ContentType:text/plain) y el estado HTTP 200 (OK).

Con res.write se responde la cadena "Hello World!" y en la línea siguiente línea se termina la petición con res.end().

En la última línea se enlaza el servidor al puerto 4000.

Así que, si ejecuta este script en Node, apunte en el navegador a http://localhost:4000 y verá la cadena “Hola Mundo" en el navegador (Teixeira, 2012, p. 55).

- Una variación de hola mundo

Para ejecutar el código es necesario grabarlo en un archivo como “server.js” por ejemplo.

Cuadro No. 2.2 CREACIÓN DE UN SERVIDOR BÁSICO "PONG"

```
var http = require('http');
var server = http.createServer(function(request, response) {
console.log("Tengo Las peticiones de encabezado:");
console.log(request.headers);
response.writeHead(200, {'Content-Type': 'text/plain'});
response.write("PONG");
response.end();
});
server.listen(8080);
```

Fuente: (Pasquali, 2013, p. 67)

Es una de las formas más básicas para la creación de un servidor. Para ejecutarlo, primero desde Node, es necesario en el navegador escribir una de las siguientes direcciones locales:

localhost:8080 ó 127.0.0.1:8080

Al ejecutarse en forma interna en la consola muestra un mensaje “Tengo las peticiones de encabezado” a continuación muestra el contenido de la petición del encabezado. Y por el lado del cliente muestra simplemente la palabra “PONG”.

- Sirviendo archivos estáticos

Si se tiene la información almacenada en el disco que se quiere que sirva como contenido web, se puede utilizar el módulo “fs”, (sistema de archivos) para cargar nuestro contenido y pasarlo a través de la llamada http.createServer. Este es un punto de partida conceptual básico para servir archivos estáticos.

Para hacer esto, se necesita algunos archivos para servir. Crear un directorio llamado “contenido”, con siguientes tres archivos:

- index.html
- style.css
- script.js

Ahora se puede visualizar el contenido del archivo index.html:

Cuadro No. 2.3 CONTENIDO DEL ARCHIVO INDEX.HTML

```
<html>

<head>
    <title>Yay Node!</title>
    <link rel=stylesheet href=styles.css type=text/css>
    <script src=script.js type=text/javascript></script>
</head>

<body>
    <span id=yay>Hola Mundo!</span> </body>

</html>
```

Fuente: (Clements, 2014, p. 13)

También se necesita el contenido del archivo style.css:

Cuadro No. 2.4 CONTENIDO DEL ARCHIVO STYLE.CSS

```
#bueno {
    font-size : 5em;
    background: blue;
    color     : yellow;
    padding   : 0.5em
}
```

Fuente: (Clements, 2014, p. 13)

Ahora el contenido del archivo script.js

Cuadro No. 2.5 CONTENIDO DEL ARCHIVO SCRIPT.JS

```
window.onload = function () {
    alert('Bueno Node!');
};
```

Fuente: (Clements, 2014, p. 13)

Y por último el código de nuestro servidor en el archivo

Cuadro No. 2.6 CONTENIDO DEL ARCHIVO SERVER.JS

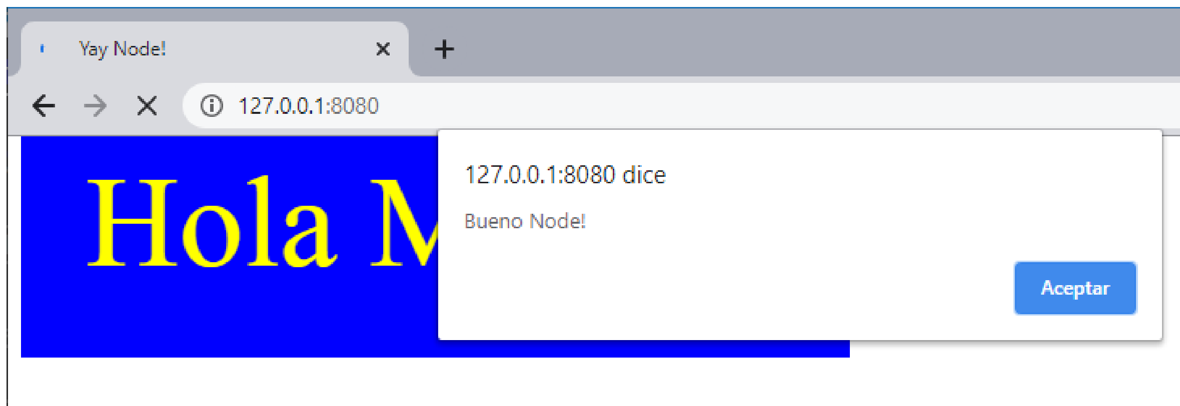
```
var http = require('http');
var path = require('path');
var fs = require('fs');
var mimeTypes = {
    '.js': 'text/javascript', '.html': 'text/html', '.css': 'text/css'
};
http.createServer(function (request, response) {
    var lookup = path.basename(decodeURI(request.url)) || 'index.html';
    var f = 'contenido/' + lookup;
    fs.exists(f, function (exists) { if (exists) {
        fs.readFile(f, function (err, data) {
            if (err) {response.writeHead(500);
                response.end('Server Error!');
                return; }
                var headers = {'Content-
type': mimeTypes[path.extname (lookup)]};
                response.writeHead(200, headers);
                response.end(data);
             });
             return;
} response.writeHead(404);
//no such file found!
response.end();
});
}).listen(8080);
```

Fuente: (Clements, 2014, p. 13)

Al ejecutar el servidor desde Node y apuntar la dirección http://localhost:4000 o http://127.0.0.1:4000

Nos presentará un resultado parecido a esto:

Figura No. 2.8 CAPTURA DE LA EJECUCIÓN DEL SERVIDOR DE ARCHIVO ESTÁTICO.



Fuente: (Clements, 2014, p. 13)

B) Stream HTTP/HTTPS Cliente

Es necesario para una aplicación de redes, hacer llamadas HTTP externas. Los servidores HTTP también suelen ser llamados a realizar servicios HTTP para los clientes que los solicitan. Node proporciona una interfaz fácil para hacer llamadas HTTP externas.

Debido al cambio de los protocolos en la actualidad, se usa para este ejemplo el módulo https, que está construido sobre la base del módulo http:

Cuadro No. 2.7 CONTENIDO DEL CÓDIGO DE UNA PETICIÓN HTTPS.

```
var https = require('https');
https.request({
    host: 'www.google.com',
    method: 'GET',
    path: "/"
}, function (response) {
    response.setEncoding("utf8");
    response.on("readable", function () {
        console.log(response.read())
    });
}).end();
```

Fuente: (Teixeira, 2012, p. 59)

El objeto HTTP no sólo proporciona capacidades de servidor, sino que también nos ofrece funcionalidad de cliente. Se podría querer usar esta funcionalidad para un sinnúmero de propósitos: APIs basadas en HTTP, recopilación desde sitios web para procesamiento estadístico o en ausencia de una API, o el primer paso en la prueba automatizada de la interfaz de usuario (Clements, 2014, p. 47).

Ahora se puede ver unas aplicaciones que hacen peticiones para consumir API´s de sitios interesantes:

- Descripción de la figura astronómica del día desde la API de la NASA

En este código, se envía una petición a la API de la NASA, y e imprime la URL de la figura astronómica del día, más bien su explicación.

Cuadro No. 2.8 CÓDIGO DE PETICIÓN HTTPS A LA API DE LA NASA

```
const https = require('https');

https.get('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY', (resp) =>
{
  let data = '';

  // Un trozo de datos ha sido recibido.
  resp.on('data', (chunk) => {
    data += chunk;
  });

  // La respuesta complete ha sido recibida. Imprime el resultado
  resp.on('end', () => {
    console.log(JSON.parse(data).explanation);
  });

}).on("error", (err) => {
  console.log("Error: " + err.message);
});
```

Fuente: (Agnew, 2020)

Variación del código que hace una descripción de la figura astronómica del día desde la API de la NASA

Se puede usar el objeto response devuelto de una llamada de http.get como nuestro stream de lectura y se establecerán algunos valores iniciales, se requerirá el módulo http como en el siguiente código:

Cuadro No. 2.9 VARIACIÓN DEL CÓDIGO FUENTE DE PETICIÓN A LA API DE LA NASA

```
var http = require('https'),
feed = 'https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY', ready = false
;
function decide(cb) {
     console.log('decidiendo');
     setTimeout(function () {
          if (Date.now()%2) {
              return console.log('solicitud rechazada');
            } ready = true; cb(); }, 2000);
}
http.get(feed, function (res) {
    res.on('readable', function log() {
    if (!ready) { return decide(log); }
    console.log(res.read()+'');
    });
  });
```

Fuente: (Agnew, 2020)

Se puede pasar la variable feed a http.get, la variable ready es usada para introducir alguna incertidumbre. Para hacer las cosas interesantes, el autor le da la habilidad al programa de escoger si quiere obedecer en hacer la petición o no a través de la función decide(cb).

Lo que se está emulando aquí, es una suerte de lógica condicional que toma un tiempo calcular. Un equivalente en el mundo real podría ser un formulario de validación o la espera de confirmación de un usuario o ambos.

La segunda parte del código descrito líneas arriba, ejecuta la petición a la API de condicionada por la variable ready (Clements, 2014, p. 128).

- Consumir una API REST falsa para pruebas y prototipos.

Ambos módulos HTTP y HTTPS están empaquetados en la biblioteca estándar. Con estos módulos, puedes hacer fácilmente una petición HTTP sin instalar paquetes externos. Pero, desafortunadamente, estos son módulos de bajo nivel y no son muy fáciles de usar en comparación con otras soluciones.

Cuadro No. 2.10 CÓDIGO DE PETICIÓN HTTPS A UNA API REST

```
const https = require('https');

https.get('https://jsonplaceholder.typicode.com/todos/2', (response) => {
  let todo = '';

  // llamado cuando un trozo de dato es recibido.
  response.on('data', (chunk) => {
    todo += chunk;
  });

  // llamado cuando una respuesta completa es recibida.
  response.on('end', () => {
    console.log(JSON.parse(todo).title);
  });

}).on("error", (error) => {
  console.log("Error: " + error.message);
});
```

Fuente: (Atta, 2019)

Los usos que se pueden dar como servidor y como cliente son ampliamente variados, existen muchas aplicaciones que se le puede dar por su amplitud de opciones en cuanto al manejo de flujos de datos. Como se puede ver en los ejemplos están siempre presentes las cadenas de datos fluyendo de subida como de bajada haciendo que la navegación por internet sea más productiva y divertida.

2.3.3.2 Streams para el sistema de archivos

Node tiene una buena API para streaming de archivos de una manera abstracta, como si fueran flujos de datos de redes, lo cual significa que se puede modelar una cantidad innumerable de aplicaciones y generar nuevas herramientas; pero a veces es necesario bajar un nivel y tratar con archivos de sistema en sí mismos.

El módulo “fs” proporciona una API para interactuar con el sistema de archivos de una manera estrechamente modelada en torno a las funciones estándar de POSIX[18] ("File System | Node.js v14.5.0 Documentation," 2020a).

El módulo “fs” maneja operaciones del sistema de archivos, tales como lectura y escritura desde archivo. Hay métodos síncronos y asíncronos en la librería.

A) Sistema de archivos síncrono y asíncrono

La lectura y escritura de datos siempre es una fuente de problemas en los programas. Las operaciones desde y hacia el disco duro son realmente lentas lo cual suele provocar que el programa no haga nada durante ese período de tiempo.

La principal idea de Node.js fue la de eliminar esa barrera y hacer que la lectura y escritura de datos en el disco no fuera tan dramático al convertirlas en tareas asíncronas. (Laguna, 2013, p. 68)

Todas las operaciones del sistema de archivos tienen formas sincrónicas y asincrónicas.

La forma asíncrona siempre toma una llamada de finalización como último argumento. Los argumentos que se pasan a la llamada de finalización dependen del método, pero el primer argumento siempre se reserva para una excepción. Si la operación se ha completado con éxito, el primer argumento será nulo o indefinido. ("File System | Node.js v14.5.0 Documentation," 2020a)

En lo ejemplos a continuación se describen operaciones asíncrona y síncrona para lectura del archivo de texto llamado entrada.txt.

[18] La Interfaz de Sistema Operativo Portable (POSIX) es una familia de estándares especificados por la Sociedad de Computación del IEEE para mantener la compatibilidad entre los sistemas operativos.

Cuadro No. 2.11 LECTURA ASÍNCRONA

```
var fs = require("fs");
// lectura asíncrona
fs.readFile('entrada.txt', function (err, data) {
if (err) {
return console.error(err);
}
console.log("Lectura asíncrona: " + data.toString());
});
```

Fuente: (Tutorials Point, 2015, p. 61).

Cuadro No. 2.12 CÓDIGO DE LECTURA DE ARCHIVO SÍNCRONO

```
// lectura síncrona
var fs = require ('fs');
var data = fs.readFileSync('entrada.txt');
console.log("Lectura síncrona: " + data.toString());
```

Fuente: (Tutorials Point, 2015, p. 61)

Los flujos de datos, acompañan el desarrollo de productos de software, desde los inicios de la era informática, lo que cambia, es la efectividad de administrarla, por ello se puede decir que se está logrando el cometido que tuvo la sugerencia de Doug McIlroy de hacer que la información deba ser entubada y manejada a manera de fluidos de datos (McIlroy, 1964).

2.4 PRUEBAS DE SOFTWARE

Una vez que se ha generado el código fuente, el software debe ser probado para descubrir y corregir, tantos errores como sea posible antes de la entrega. Su objetivo es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores, ahí es donde las técnicas de prueba de software entran en escena. Estas técnicas proporcionan una guía sistemática para diseñar pruebas que (1) ejerciten la lógica interna y las interfaces de cada componente del software y (2) ejerciten los dominios de entrada y salida del programa para descubrir errores en la función, el comportamiento y el rendimiento del programa. Las pruebas presentan un interesante dilema para los ingenieros de software, que por su

naturaleza son personas constructivas. Las pruebas requieren que el desarrollador descarte las nociones preconcebidas de la "corrección" del software recién desarrollado y luego trabaje duro para diseñar casos de prueba para "romper" el software.

Durante las primeras etapas de las pruebas, un ingeniero de software realiza todas las pruebas. Sin embargo, a medida que el proceso de prueba progresa, los especialistas en pruebas pueden involucrarse.

2.4.1 Prueba de Caja Negra

Entre los tipos de las pruebas de software está la prueba de software de caja negra donde es más importante afrontar las entradas y las salidas al sistema.

Las pruebas de caja negra intentan encontrar errores en las siguientes categorías: (1) funciones incorrectas o faltantes, (2) errores de interfaz, (3) errores en las estructuras de datos o en el acceso a bases de datos externas, (4) errores de comportamiento o de rendimiento, y (5) errores de inicialización y terminación. (Pressman y Maxim, 2015, p. 510)

Las pruebas de liberación son normalmente un proceso de pruebas de caja negra en el que las pruebas se derivan de la especificación del sistema. El sistema se trata como una caja negra cuyo comportamiento sólo puede determinarse estudiando sus entradas y las salidas relacionadas. Otro nombre para esto es prueba funcional, llamada así porque el probador sólo se ocupa de la funcionalidad y no de la implementación del software. (Sommerville, 2016, p. 246)

Las pruebas de caja negra, también llamadas pruebas de comportamiento o pruebas funcionales, se centran en los requisitos funcionales del software. Es decir, las técnicas de prueba de caja negra permiten derivar conjuntos de condiciones de entrada que ejercitarán plenamente todos los requisitos funcionales de un programa. Las pruebas de caja negra no son una alternativa a las técnicas de caja blanca. Más bien es un enfoque complementario que probablemente descubra una clase diferente de errores que los métodos de caja blanca. (Pressman y Maxim, 2015, p. 510)

CAPITULO III

MARCO APLICATIVO

CAPITULO III

3. MARCO APLICATIVO

3.1 DESCRIPCIÓN DEL CONTEXTO EN CONTROL DEL DESARROLLO Y PRODUCCIÓN DE SOFTWARE

En este sentido no se puede afirmar ni asegurar que tales controles existan, pero al menos después de revisar repositorios de universidades importantes del ámbito local no se pudo encontrar investigaciones con estrecha relación en este contexto.

En el ámbito empresarial tampoco se puede afirmar o negar la existencia de tales controles, ya que las nuevas metodologías de desarrollo ágiles van ganando cada vez mayor espacio en donde son equipos de desarrollo que combinan esfuerzos para producir software y el mantenimiento necesario para ello.

3.2 REQUERIMIENTOS FUNCIONALES Y NO FUNCIONALES

3.2.1 Requerimientos funcionales

3.2.1.1 Métricas de complejidad ciclomática

- Obtención del valor de la complejidad ciclomática del código fuente de cada procedimiento stream.

3.2.1.2 Métricas de rendimiento

- Registro del tiempo de proceso de preparación de módulos necesarios en la ejecución del código fuente de cada procedimiento stream.
- Registro del tiempo de ejecución del código fuente de cada procedimiento stream.
- Representación gráfica en el eje de coordenadas del código fuente del tiempo de proceso de preparación de módulos necesarios en la ejecución del procedimiento stream.
- Representación gráfica en el eje de coordenadas del tiempo de ejecución del procedimiento stream.

3.2.1.3 Prueba de software de caja negra

- Alternancia de entrada del formato de archivo de video para la ejecución de las pruebas del código fuente de cada procedimiento stream.

3.2.1.4 Medición del grado confiabilidad

- Obtención del grado de confiabilidad del código fuente de cada procedimiento stream.
- Obtención de diferencia de medias del grado de confiabilidad de procedimientos stream.

3.2.2 Requerimientos no funcionales

3.2.2.1 Adaptabilidad

- Capacidad de integración del módulo para medición de rendimiento de Node.Js, en diferentes plataformas de desarrollo y producción Unix, Windows, Mac.

3.3 DISEÑO CON METODOLOGÍA EXPERIMENTAL

3.3.1 Datos Iniciales

3.3.1.1 Planteamiento general del problema

El desarrollo y producción de software, viene acompañado en algunos casos por la falta de aplicación de técnicas de administración y control, que pueden derivar en daños o perjuicios leves en algunos casos o muy serios y con graves consecuencias en otros dependiendo del tipo de sistema y el contexto en el que se vaya a usar.

3.3.1.2 Hipótesis

Determinar si existe diferencia significativa, entre los grados de confiabilidad de procedimientos stream basado en su complejidad ciclomática para coadyuvar a la toma de decisiones adecuada para asegurar la calidad del producto de software en Node.js.

3.3.1.3 Objetivo

Aplicar técnicas de medición y métricas para determinar si existe diferencia entre los grados de confiabilidad de procedimientos stream en base a su complejidad ciclomática para coadyuvar al aseguramiento de la calidad del producto de software en Node.js para la toma de decisiones correspondientes.

3.3.1.4 Zona Geográfica

Por tratarse de un tema socio técnico, se encuentra ubicado espacialmente en cualquier lugar donde se encuentre en funcionamiento el sistema expuesto a la técnica de medición y métricas que se propone en esta investigación dependiente del dispositivo y sus recursos de hardware como ser microprocesador, memoria, dispositivos de entrada y salida.

3.3.1.5 Recursos de Investigación

Tabla No. 3.3 RECURSOS DE INVESTIGACIÓN

Tipo	Categoría	Recurso	Descripción
Recursos bibliográficos	Material	Bibliografía	Libros, revistas científicas y documentos electrónicos en formato de documento portable.
Recursos tecnológicos	Equipamiento	Equipo	Microprocesador Core i3, ram 8 Gb, SSD 250Gb, plataforma windows 10
		internet	Fibra óptica 1 mb bajada, 3 mb de subica
		Lenguaje del tipo Servidor	JavaScript Node versión 14.15.1 LTS
		API de medición de rendimiento	Modulo nativo performance hook correspondiente a la medición de rendimiento incluido en Node.js
	Repositorio	Publicación de proyectos	https://github.com/ es un sitio que permite alojamiento para proyectos y sus bifurcaciones a través de git[19].

Fuente: (Elaboración propia)

3.3.1.6 Precisión

El módulo nativo de rendimiento que provee Node.js, provee una implementación de un subconjunto de la API[20] de rendimiento web, con especificaciones de la W3C.

[19] Sistema local de control de versiones de proyectos, trabaja en combinación con el sitio https://github.com/

[20] Interfaz de programación de aplicaciones, que es un medio de comunicación entre sistemas.

Todo ello con registros de tiempo de alta resolución que no está sujeto a ajustes o desviaciones del reloj del sistema.

3.3.2 Muestreo

3.3.2.1 Selección de usuarios y códigos fuente para su medición

Por las características de la investigación se selecciona del repositorio de proyectos del sitio github en base a algunos proyectos que corresponden a diferentes autores con sus variaciones en la complejidad ciclomática de procedimientos stream:

Tabla No. 3.4 SELECCIÓN DE CÓDIGO FUENTE EN EL REPOSITORIO DE PROYECTOS GITHUB

No.	Link de usuario en el repositorio de github	Usuario	Complejidad Ciclomática del código fuente[21]
1	https://gist.github.com/paolorossi/1993068	Paolorossi	3
2	https://github.com/daspinola/video-stream-sample	daspinola	4

Fuente: (Elaboración propia)

3.3.3 Instrumentos de Investigación

3.3.3.1 Marcadores de rendimiento de node.js.

Estos marcadores de rendimiento, son colocados dentro del código en lugares estratégicos y específicos que registran el tiempo en que los eventos suceden.

3.3.3.2 Capturas de Excepciones en el Código Fuente

Los lenguajes de programación y los entornos de programación como JavaScript cuentas con sentencias que permiten capturar las excepciones al código fuente, dicho de otra manera, capturan el error y permiten administrarlo, es en el momento de esa captura de error cuando se introduce marcadores de rendimiento para capturar el instante del error.

[21] https://www.npmjs.com/package/complexity-report herramienta para análisis de complejidad de software que incluye la métrica de complejidad ciclomática.

3.3.3.3 Prueba de caja negra

Se alterna la entrada de archivos de video al sistema o código fuente, para alimentar el procedimiento stream en turno de prueba con los siguientes archivos que varían en tamaño y formato de video que sirve para realizar el cálculo del muestreo en el código fuente y generar algún posible error, es necesario aclarar que el contenido o título del video es indiferente.

Tabla No. 3.5 TAMAÑO Y FORMATO DE ARCHIVOS PARA ALIMENTAR LA CAJA NEGRA

No.	Tamaño KB[22]	Formato de video	Descripción del formato de video	Tiempo de duración
1	15.858	MP4	Basado en formato de Quicktime, eficiente en compresión de datos y alta definición de video	00:01:31
2	13.417	WMV	Microsoft es propietario de este formato y tiene soporte para gestión digital de derechos	00:12:35
3	3073	MP4	Basado en formato de Quicktime, eficiente en compresión de datos y alta definición de video	00:01:30
4	375	MP4	Basado en formato de Quicktime, eficiente en compresión de datos y alta definición de video	00:00:05
5	428	OGG	Es un contenedor multimedia, diseñado como alternativa al AVI	00:00:05
6	225	WEBM	Formato contenedor para video y audio, fue ideado para uso con HTML5	00:00:05
7	6.671	MP4	Basado en formato de Quicktime, eficiente en compresión de datos y alta definición de video	00:00:38

fuente: (Elaboración propia)

3.4 DESARROLLO DE LA TÉCNICA DE MEDICIÓN DE CONFIABILIDAD

3.4.1 Procesamiento de datos

Se realizó en forma automática, colocando marcadores de rendimiento como en el Cuadro No. 3.1, recopilando datos en formato JSON[23] en un archivo de texto como

[22] Unidad de medida de la información en computadoras, cada KB es equivalente a 1024 Bytes.

[23] Notación de objetos de JavaScript, es un estándar que es muy usado en el ámbito web para el intercambio de información.

en el Cuadro No. 3.2, con ayuda de los marcadores de rendimiento registran el rendimiento de cada proceso, que después es interpretado por el navegador para su representación gráfica en el eje de coordenadas en formato de líneas como en el Gráfico No. 3.1.

Cuadro No. 3.1 EJEMPLO DE CÓDIGO FUENTE CON APLICACIÓN DE MARCADORES DE RENDIMIENTO

```
//llamada al módulo para realizar medición de rendimiento
const { PerformanceObserver, performance } = require('perf_hooks');
const obs = new PerformanceObserver((items) => {
    console.log('Observación realizada',items.getEntries());
    performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });
let total=0;
// Definición de función de prueba para sumar 2 números
function suma(a,b){return a+b};
 // ********PRIMER MARCADOR DE RENDIMIENTO
performance.mark('A');
total = suma(100,200);
// *********SEGUNDO MARCADOR DE RENDIMIENTO
performance.mark('B');
// sentencia para realizar la medición del segmento A hasta B
performance.measure('Medición primer segmento', 'A', 'B');
console.log(`Suma ${total}`);
// como resultado se obtiene la siguiente salida en consola:
//
// Observación realizada [
//   PerformanceEntry {
//     name: 'Medición primer segmento',
//     entryType: 'measure',
//     startTime: 37.4458,
//     duration: 0.0265 -→ ES EL TIEMPO DE RENDIMIENTO EN LA FUNCION SUMA
//   }
// ]
// Suma 300
```

Fuente: (Elaboración propia)

El código fuente anterior es un ejemplo sobre la forma en que se puede capturar marcas de rendimiento en los lugares o segmentos de código fuente. Además, se

usó la captura de las entradas fallidas al sistema que también fueron registradas para realizar el conteo de las ocurrencias de éxito y falla.

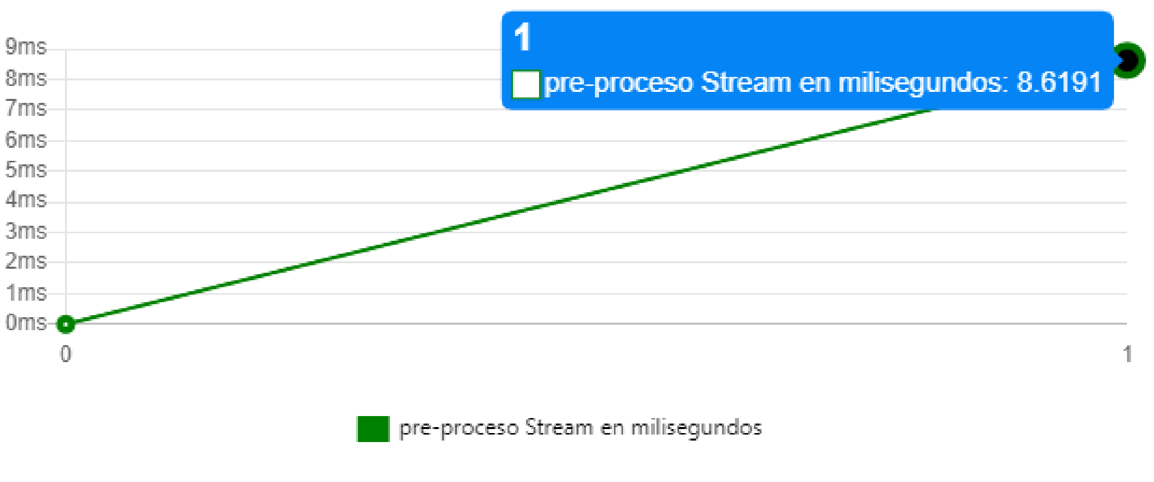
Cuadro No. 3.2 EJEMPLO DE CONTENIDO DEL ARCHIVO DE REGISTROS EN NOTACIÓN DE OBJETOS JAVASCRIPT UN ARCHIVO CON DIFERENTES REGISTROS


```json
[
    {
        "name": "Pre-proceso",
        "entryType": "measure",
        "startTime": 30.1828,
        "duration": 8.6191
    },
    {
        "name": "Proceso_stream",
        "entryType": "measure",
        "startTime": 38.8019,
        "duration": 10208.573801
    },
    {
        "name": "Error",
        "entryType": "measure",
        "startTime": 29.2092,
        "duration": 7.2389
    }
]
```


Fuente: (Elaboración Propia)

Luego este archivo es procesado para obtener una gráfica en el eje de coordenadas con esos registros de tiempo para ser presentado en un servidor local como se puede apreciar en el siguiente gráfico.

Gráfico No. 3.1 EJEMPLO DE GRAFICO DEL TIPO LINEA, EN EL SERVIDOR LOCAL CON CHART.JS[24]



Fuente: (Elaboración propia)

3.4.2 Método de Análisis

Para el análisis de datos, se usó la estadística descriptiva en el caso de las muestras de rendimiento de cada proceso.

Para la prueba de hipótesis se usó la denominada diferencia de medias para muestras grandes y varianza conocida, sobre los registros de marcas de tiempo en milisegundos ver Cuadro No. 3.2, obtenidas por el archivo en notación de objetos JavaScript, para comprobar si la diferencia observada entre las dos medias muéstrales es estadísticamente significativa.

Para mayor comodidad del análisis de datos se hizo la importación de los registros de las marcas de rendimiento a la planilla de cálculo Excel.

[24] https://www.chartjs.org/ Herramienta de uso libre para desarrolladores y diseñadores web.

3.4.3 Análisis de Datos

3.4.3.1 Resultados de las demandas hechas al Sistema de Pruebas de Código Fuente

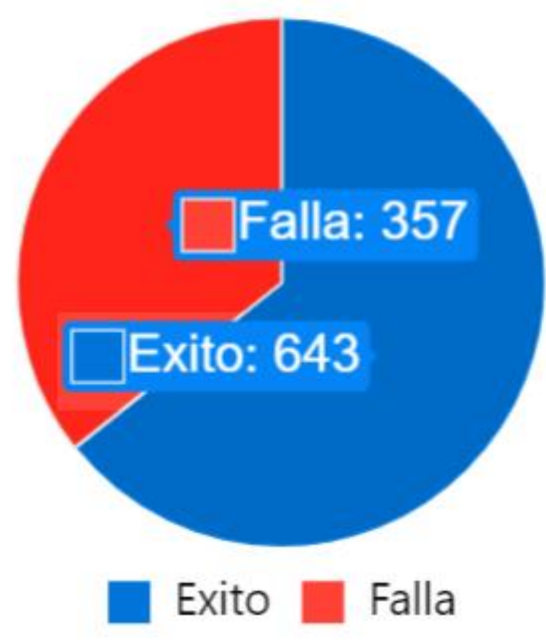
Tabla No. 3.6 RESUMEN DE PRUEBAS DE CÓDIGO FUENTE

Código fuente de Usuario	Complejidad Ciclomática	Entradas al sistema de prueba		Total pruebas por usuario
		Éxito	Falla	
daspinola	4	1000	0	1000
paolorossi	3	643	357	1000

Fuente: (Elaboración propia)

3.4.3.2 Descripción de los Resultados del Sistema de Pruebas

Gráfico No. 3.2 CAPTURA DEL REPORTE DE RENDIMIENTO DE CÓDIGO FUENTE DEL USUARIO PAOLOROSSI



Fuente: (Elaboración propia)

La gráfica anterior, corresponde a la captura del reporte de rendimiento del código fuente del usuario paolorossi de la página github, que representa una probabilidad

de falla a demanda PDFAD igual a 357/1000, lo que equivale a decir que de 1000 entradas al sistema 357 entradas probablemente fallarán con las condiciones con las que se hizo la prueba. Por otro lado, la gráfica también indica que de 1000 entradas al sistema 643 probablemente tendrán éxito.

Gráfico No. 3.3 CAPTURA DEL REPORTE DE RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO DASPINOLA

Probabilidad de Falla a Demanda: daspinola

Fuente: (Elaboración propia)

La gráfica anterior, corresponde a otra captura del reporte de rendimiento del código fuente del usuario daspinola, de la página github, que no representa ninguna probabilidad de falla a demanda PDFAD, lo que equivale a decir que de 1000 entradas al sistema ninguna entrada probablemente fallará en las condiciones con las que se hizo la prueba del sistema. En ese mismo sentido también se puede indicar que probablemente el sistema tendrá 1000 entradas exitosas.

3.4.3.3 Resultados del Sistema de Pruebas sobre el Rendimiento

Las marcas de rendimiento son registros de tiempo en milisegundos, en notación de objetos de JavaScript que son leídas del registro para generar un reporte en el servidor local.

Figura No. 3.1 CAPTURA DEL REPORTE DE RESULTADOS SOBRE EL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO PAOLOROSSI

Resumen :

Prueba del código fuente: Paolo Rossi

Suma: 1463292.1624999987
Promedio: 1464.2921624999988
No. de Registros: 1000
Entradas exitosas: 643
Entradas fallidas: 357

Fuente: (Elaboración propia)

Esta es la captura de servidor local, con los resultados sobre el rendimiento de código fuente del usuario paolorossi que contiene 357 entradas fallidas, 643 entradas exitosas y un promedio de entrada de 1464.292 milisegundos sobre 1000 registros u operaciones del sistema.

Figura No. 3.2 CAPTURA DEL REPORTE DE RESULTADOS SOBRE EL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO DASPINOLA

Resumen:

Prueba del código fuente: daspinola

Suma: 1661597.5510999986
Promedio: 1662.5975510999986
No. de Registros: 1000
Entradas exitosas: 1000
Entradas fallidas: 0

Fuente: (Elaboración propia)

Esta es la captura de servidor local, con los resultados sobre el rendimiento de código fuente del usuario paolorossi que contiene 1000 entradas fallidas, 0entradas exitosas y un promedio de entrada de 1662.597 milisegundos sobre 1000 registros u operaciones del sistema.

3.4.3.4 Tablas de Frecuencias en Excel Sobre Resultados del Rendimiento Registrado.

Para tener mayor flexibilidad y comodidad de manipulación de datos, se hizo la exportación de los datos correspondientes al rendimiento con un pequeño programa para generar un archivo de texto que luego será importado por la planilla de cálculo en Excel, sobre el dato correspondiente a las marcas de tiempo.

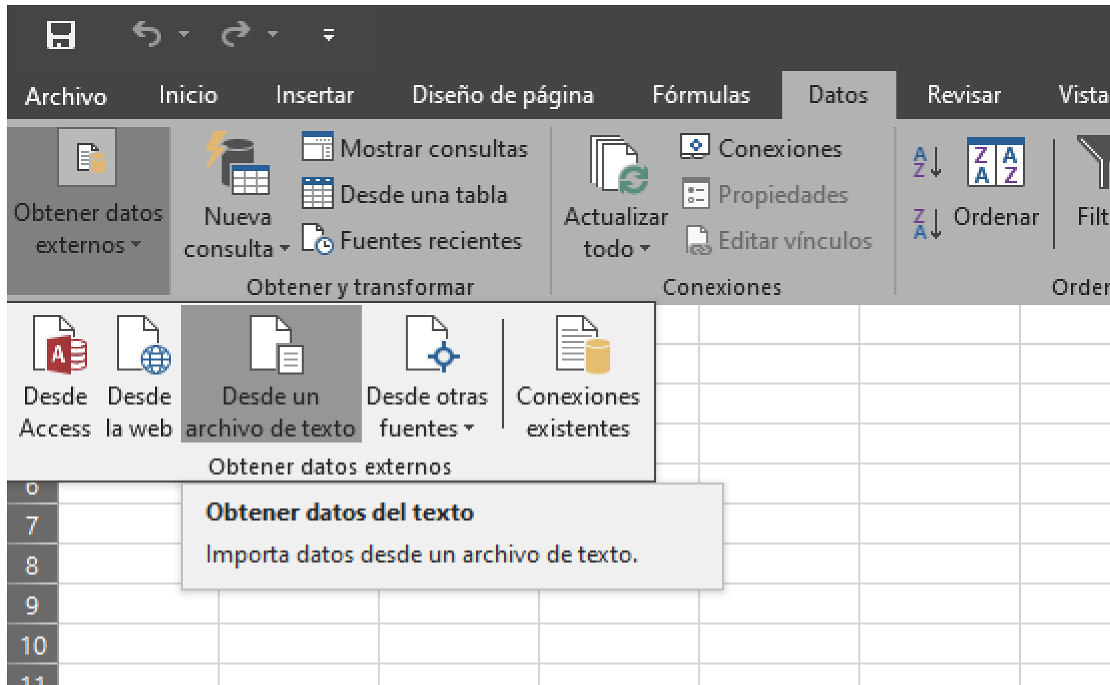
Cuadro No. 3.3 PROGRAMA AUXILIAR PARA EXPORTACIÓN DE DATOS DE MARCAS DE TIEMPO A FORMATO TEXTO, SEPARADO POR COMAS

```
const fs= require('fs')
const file = require('./daspinola.json')

// console.log(file)
const proceso=file.filter(elt=>elt.name=="Proceso_stream").map((elt)=>elt
.duration)
const error=file.filter(elt=>elt.name=="Error").map((elt)=>elt.duration)
const completo = proceso.concat(error)
fs.writeFile('descriptivaDaspinola.txt',JSON.stringify(completo),(err)=>{
    if (err) throw err;
    console.log ('Grabado')
})
```

Fuente: (Elaboración propia)

El anterior cuadro carga el archivo en notación de objetos JavaScript, filtra los datos correspondientes al rendimiento de la duración del tiempo de proceso y escribe los datos filtrados en un archivo de texto. Ese archivo de texto generado fue importado desde la planilla de cálculo Excel de la manera que se puede apreciar en el siguiente gráfico.

Figura No. 3.3 HERRAMIENTA PARA IMPORTAR DATOS EXTERNOS A LA PLANILLA DE CALCULO EXCEL



Fuente: (Elaboración propia)

Una vez que se hizo la importación de los 1000 registros de marcas de tiempo en milisegundos, para cada prueba de rendimiento, se hizo la agrupación de datos en tablas de frecuencias para poder representar ambos grupos en dos tablas con la ayuda de la planilla de cálculo Excel y las funciones estadísticas que se encuentran disponibles.

Para la construcción de las tablas se usó la regla de Sturges para el cálculo del intervalo de clases.

$$k = 1 + 3.322 * log_{10}(N)$$

Tabla No. 3.7 DATOS DE LA TABLA DE FRECUENCIAS DE LA PRUEBA DEL CÓDIGO DEL USUARIO DASPINOLA

Promedio	1654
Desviación estandar	1228
Máximo	20632
Mínimo	949
Rango	19683
Coeficiente de Variación	74%
No. de datos	1000
Intervalo de clases	11
Ancho de clase	1789

Fuente: (Elaboración propia)

Tabla No. 3.8 TABLA DE FRECUENCIAS DEL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO: DASPINOLA

No.	Límite Inferior	Límite Superior	Frecuencia (fi)
1	0	1789	842
2	1789	3578	130
3	3578	5367	12
4	5367	7156	6
5	7156	8945	3
6	8945	10734	3
7	10734	12523	1
8	12523	14312	1
9	14312	16101	1
10	16101	17890	0
11	17890	20632	1
			1000

Fuente: (Elaboración propia)

Tabla No. 3.9 DATOS DE LA TABLA DE FRECUENCIAS DE LA PRUEBA DEL CÓDIGO DEL USUARIO PAOLOROSSI

Promedio	1465
Desviación estandar	1010
Maximo	8480
Minimo	34
Rango	8447
Coeficiente de Variación	69%
No. De datos	1000
Intérvalo o de clases	11
Ancho de clase	768

Fuente 1 (Elaboración propia)

Tabla No. 3.10 TABLA DE FRECUENCIAS DEL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO PAOLOROSSI

No.	Límite Inferior	Límite Superior	Frecuencia (fi)
1	0	768	264
2	768	1536	219
3	1536	2304	396
4	2304	3072	67
5	3072	3840	31
6	3840	4608	13
7	4608	5376	4
8	5376	6144	1
9	6144	6912	3
10	6912	7680	0
11	7680	8480	1
			999

Fuente 2 (Elaboración propia)

3.4.3.5 Prueba de Hipótesis

Tomando el promedio de rendimiento de las tablas anteriores, de dos procesos con muestras iguales y varianzas conocidas:

$$\mu_A = Promedio\ de\ rendimiento\ del\ código\ fuente\ del\ usuario\ paolorossi$$

$$\mu_B = Promedio\ de\ rendimiento\ del\ código\ fuente\ del\ usuario\ daspinola$$

1. Se propone las siguientes hipótesis

$$H_0: \mu_A = \mu_B \Rightarrow \mu_A - \mu_B = 0$$

$$H_1: \mu_B > \mu_A \Rightarrow \mu_B - \mu_A > 0$$

2. Se define la significación: $\alpha = 0{,}05$

3. Se calcula los valores críticos y de prueba
   a. Como la muestra es grande es valor crítico es obtenido según tablas de referencia:

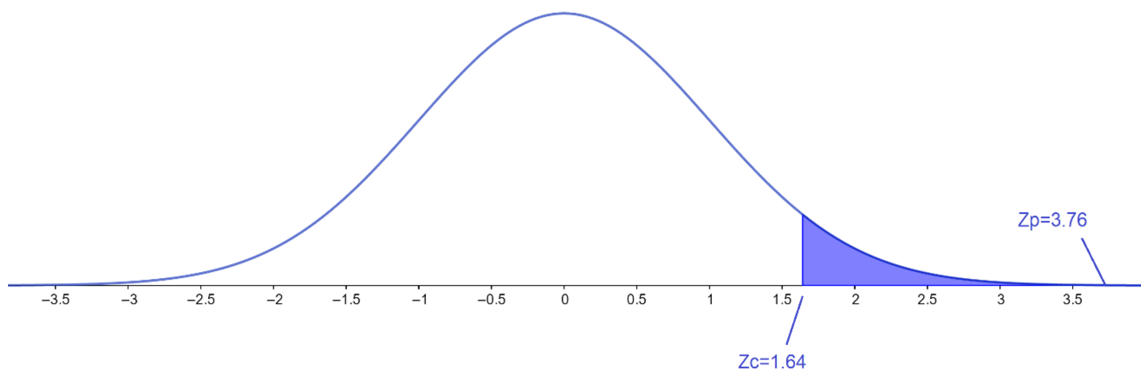
$$Z_c = 1.64$$

   b. Para el valor de prueba se usó:

$$Z_p = \frac{\overline{x_1} - \overline{x}_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

$$Z_p = \frac{1654 - 1465}{\sqrt{\frac{1228^2}{1000} + \frac{1010^2}{1000}}} = 3.76$$

Gráfico No. 3.4 VALORES CRITICO Y DE PRUEBA EN LA DISTRIBUCIÓN NORMAL



Fuente 3 (Elaboración propia)

4. Decisión y conclusión

En base a la comparación de los valores crítico y de prueba se hizo la siguiente apreciación.

- Decisión: Se rechaza la hipótesis nula H_0 y se acepta la hipótesis alternativa H_1, ya que el valor de Zp prueba obtenido es mayor que el valor crítico Zc, entrando en una zona de rechazo en la gráfica de distribución normal de la hipótesis H_0.
- Conclusión: El promedio de rendimiento del código fuente del usuario daspinola es mayor al promedio de rendimiento del código fuente del usuario paolorossi con una significación del 5%.

3.4.4 Informe Final

3.4.4.1 Resumen

Dados dos sujetos de prueba en este caso, será el código fuente de dos usuarios de la página de github, acerca de procedimientos stream, sometidos al proceso de prueba de caja negra, se obtiene la probabilidad de falla a demanda y se somete a prueba de hipótesis las medias de las muestras de rendimiento de ambos procedimientos para afirmar o rechazar si existe diferencia o no acerca de ellas.

3.4.4.2 Probabilidad de falla a Demanda PDFAD

Por la información brindada por el sistema de pruebas, se puede asegurar que el rendimiento del código fuente del usuario de github daspinola, tiene el mejor rendimiento en las condiciones y con las pruebas realizadas.

Tabla No. 3.11 PROBABILIDAD DE FALLA A DEMANDA DE LOS SUJETOS DE PRUEBA

Codigo de usuario	Complejidad Ciclomática	Probabilidades en 1000 demandas	
		Éxito	Fracaso
daspinola	4	100%	0%
paolorossi	3	64,30%	35,70%

Fuente: (Elaboración propia)

3.4.4.3 Diferencia de Medias de Rendimiento

Por otro lado, del resultado de la prueba de hipótesis del valor en tablas:

$$Z_c = 1.64$$

Y el resultado del valor de prueba:

$$Z_p = 3.76$$

se observa que Zp es mayor que Zc, por lo cual para la prueba de hipótesis planteada H_0, queda rechazada y se acepta H_1 para la media de rendimiento del código fuente del usuario daspinola, que es superior al rendimiento del código fuente del usuario paolorossi.

Por lo cual es posible recomendar, se pueda tomar en cuenta estos resultados para asumir las decisiones en cuanto a la implementación de esos procedimientos en evaluación.

3.4.4.4 Discusión

Los resultados obtenidos reflejan el procedimiento en las condiciones con las que se planeó las pruebas de rendimiento con las entradas de datos, el hardware en donde radica el entorno del software usado. Estos resultados tienen una validez en las condiciones de desarrollo y para las condiciones de producción, se requeriría planear el mismo procedimiento y validar los resultados y compararlos.

CAPITULO IV

CALIDAD Y SEGURIDAD

CAPITULO IV

4. CALIDAD Y SEGURIDAD

4.1 OBTENCIÓN DE RESULTADOS DE CALIDAD

Para obtener resultados de calidad del módulo nativo de Node.js como instrumento para esta investigación para medir el rendimiento del código fuente en procedimientos stream, se tiene la seguridad que la herramienta está diseñada siguiendo las especificaciones definidas por la W3C acerca del origen del tiempo y el tiempo actual con una resolución de sub-milisegundos, tal que no está sujeta a las variaciones o ajustes del reloj del sistema.

4.2 CICLO DE VIDA DEL MODELO DE PRUEBAS

El modelo, depende de la necesidad de controlar el desarrollo y producción de software, en este caso de investigación de la confiabilidad y la información que necesite conocer aspectos como las transacciones y rendimiento de su código fuente, bajo la siguiente posible secuencia de procedimientos:

1. Selección de código fuente, aplicando la métrica de complejidad ciclomática para obtener una referencia acerca de la complejidad de diseño de los, sobre los cuales exista la posibilidad de introducir marcadores de rendimiento para capturar el tiempo en que los eventos de los cuales se necesita conocer.
2. Selección de todos los posibles datos o archivos que según los requerimientos del código fuente deban ser procesados por esa porción de código, esto también es conocido como pruebas de caja negra donde es más importante los tipos de entrada y salida del sistema.
3. Realizar el ciclo de pruebas prototipo para asegurar la precisión de los datos que serán recolectados por los marcadores de rendimientos introducidos en el código fuente.
4. Inicio del ciclo de pruebas aprobado para recolectar las muestras de los datos de rendimiento y demandas al sistema.
5. Escoger un modelo de análisis datos
6. Analizar los datos

7. Presentación de informe

4.2.1 Estimación de Pagos al Personal

Para la estimación de los pagos referentes a las personas involucradas en la implementación se hace la estimación como en la tabla siguiente.

Tabla No. 4.12 CALCULO DE PLANILLA DE PAGOS AL PERSONAL INVOLUCRADO EN EL PROYECTO

(en bolivianos)

Paso No.	Personal requerido	Cantidad de personal requerido	Tiempo de trabajo (días)	Salarios mínimos para el cálculo *	Haber diario	Pago total
1,2,3,4,5,6,7	Ingeniero de Software/Sistemas	1	60	4	283	16976
4	Tester de calidad de software	1	5	2	141	707
				Total a pagar		17683

* Salario mínimo nacional en Bolivia 2122

Fuente 4 (Elaboración propia)

4.2.2 Tiempo Estimado de Trabajo

Analizando la anterior tabla, es posible estimar el tiempo de trabajo en 65 días hábiles de trabajo, al inicio el ingeniero de sistemas o software elabora el prototipo hasta el paso 3 del ciclo de vida del proyecto mencionado antes. Después pasa a responsabilidad del tester[25] calidad de software quien entrega los datos recolectados de las pruebas del sistema para ser analizados y para elaborar el informe correspondiente.

[25] Personal técnico encargado de realizar las pruebas de software

CAPITULO V

COSTO BENEFICIO

CAPITULO V

5. COSTO BENEFICIO

5.1 COSTOS

Haciendo una relación de ingresos, se elabora una tabla haciendo una proyección de egresos e ingresos en un periodo de tiempo de 12 meses, para estimar un índice de costo beneficio por la tarea de evaluar aspectos de calidad de proyectos de software, de acuerdo a la siguiente tabla:

Tabla No. 5.13 ESTIMACIÓN DE COSTO BENEFICIO EN UN PERIODO DE 12 MESES

(Expresado en bolivianos)

		Meses	12
Egresos			
Descripción	Cantidad	Unitario	Estimación de costo
Activos Fijos			
Equipo de computación	1	5000	5000
Impresora	1	1200	1200
Gastos corrientes			
Honorarios profesionales	13	7780	101140
Salario secretaria	13	2122	27586
Alquiler oficina	12	1400	16800
pago de servicios agua y luz	12	30	360
Conexión a internet	12	160	1920
		Suma egresos	154006
Ingresos			
Costo por proyecto	24	11670	280080
		Suma ingresos	280080
		Diferencia	126074

Fuente: (Elaboración propia)

5.2 BENEFICIO

De la tabla anterior se puede aclarar el costo por proyecto en la columna descripción asciende a un monto de Bs. 11670 que resulta de añadir un 50% a los honorarios profesionales, vale decir Bs.7780 + Bs. 3890.

Por otro lado, el índice de costo beneficio benefició resultará de dividir beneficios entre costos:

$$I = B/C$$

$$I = 126074/154006$$

$$I = 1{,}81$$

Lo que podría significar que por cada boliviano invertido se tendría un retorno de Bs. 1,81.

CAPITULO VI

CONCLUSIONES Y RECOMENDACIONES

CAPITULO VI

6. CONCLUSIONES Y RECOMENDACIONES

6.1 CONCLUSIONES

- Hacer uso de las técnicas estadísticas para comparar y medir la diferencia de los grados de confiabilidad de procedimientos es stream, resulta ser una forma ideal, cuando no se cuenta con otros datos como ser tiempos medios para fallar, tiempo medio entre fallos, tiempo medio de uso, tiempo medio de reparación.
- Las herramientas nativas que ofrece el entorno de JavaScript Node, para la implementación de métricas de rendimiento, ofrecen una precisión de datos importante y siguen las recomendaciones de la W3C, dado a que no utiliza los recursos de reloj del dispositivo donde se encuentra operando, sino paralelamente inicia un reloj denominado monotónico que opera desde el inicio de la invocación del proceso en curso. Como muchos autores recomiendan, que la elección de los instrumentos usados para la recolección de datos y su precisión, es muy importante, la mencionada herramienta de rendimiento usada para este proyecto, cumple con esas condiciones recomendadas, tanto así que ningún dato de rendimiento de las 1000 transacciones se repite.
- Conocer el valor de la complejidad ciclomática, que muestra algunos rasgos importantes de diseño del código fuente en procedimientos de software, resulta ser de utilidad ya que, sin profundizar mucho en los mecanismos usados por el programador, en el código fuente, proveen de un panorama general de las condiciones de complejidad del código y en esta investigación se usó para comparar la homogeneidad del código fuente en observación experimental.

6.2 RECOMENDACIONES

Es de mucha importancia a veces vital, observar y medir el comportamiento de los sistemas, asimismo implementar metodologías distintas para obtener mayor

amplitud y ampliar el conocimiento acerca de las técnicas de medición y métricas de software, también es importante seguir aprendiendo de la teoría de confiabilidad y rescatar todo aquello que la investigación en la industria avanzó.

Asimismo, se invita a todas las personas involucradas en el desarrollo y la producción de software continuar con este tipo de investigaciones, ya que si no se conoce el comportamiento de los sistemas y datos que se encuentran inscritos en su interior difícilmente se puede realizar mejoras al mismo.

En las aplicaciones que son lanzadas muy rápido gracias a la evolución de las comunicaciones en internet en la sociedad, es importante ofrecer productos de software confiable y actuar con responsabilidad social hacia la población que merece el respeto necesario, de los profesionales en el contexto social.

BIBLIOGRAFÍA

Abran, A. (2010). *Software metrics and software metrology*. Hoboken N.J., Los Alamitos CA: Wiley; IEEE Computer Society.

Agnew, S. (2020). *5 Ways to Make HTTP Requests in Node.js*. Recuperado de https://www.twilio.com/blog/2017/08/http-requests-in-node-js.html

Atta. (2019). *7 Ways to Make HTTP Requests in Node.js*. Recuperado de https://attacomsian.com/blog/http-requests-in-nodejs/

Brown, E. (2014). *Web development with Node and Express* (First edition). Beijing, Sebastopol CA: O'Reilly.

Clements, D. M. (2014). *Node cookbook: Over 50 recipes to master the art of asynchronous server-side JavaScript using Node.js, with coverage of Express 4 and Socket.IO frameworks and the new Streams API* (2nd ed.). *Quick answers to common problems*. Birmingham, UK: Packt Pub. Recuperado de http://proquest.tech.safaribooksonline.de/9781783280438

Cuevas Callisaya, B. (2006). *Modelo de Ingeniería del Mantenimiento para Productos Software Orientado a Objetos*. Universidad Mayor de San Andrés (La Paz - Bolivia).

Fenton, N. E. y Bieman, J. (2014). *Software metrics: A rigorous and practical approach* (Third edition). *Chapman & Hall/CRC innovations in software engineering and software development*. Boca Raton: CRC Press, Taylor & Francie Group.

File System | Node.js v14.5.0 Documentation. (2020a). Recuperado de https://nodejs.org/api/fs.html

Gallaba, K. (2015). *Characterizing and refactoring asynchronous JavaScript callbacks*. The University of British Columbia (Vancouver).

HTTP | Node.js v14.5.0 Documentation. (2020b). Recuperado de https://nodejs.org/api/http.html

Huanca Aliaga, H. B. (2011). *Elaboración de Métricas para Evaluar la Seguridad del Software Durante su Desarrollo*. Universidad Mayor de San Andrés (La Paz - Bolivia).

Jansson, L. (2017). Parallelism in Node.js applications [Paralelismo en Aplicaciones Node.js]: Data flow analysis of concurrent scripts [Análisis de Flujos de Datos de Scripts Concurrentes].

Laguna, A. (2013). *Descubriendo Node.js y Express: Aprender a desarrollar en Node.js y descubre cómo aprovechar Express.*

Laird, L. M. y Brennan, M. C. (2006). *Software measurement and estimation [Medición y Estimación de Software]: A practical approach [Una Aproximación práctica]*. Hoboken N.J.: John Wiley & Sons.

Mardan, A. (2018). *Practical Node.js*. Berkeley, CA: Apress.

McIlroy, D. (1964). *Prophetic Petroglyphs.*

Mili, A. y Tchier, F. (2015). *Software testing: Concepts and operations*. Hoboken New Jersey: Wiley.

Muñoz Razo, C. (2011). *Cómo elaborar y asesorar una investigación de Tesis* (Segunda Edición): Pearson.

Nicolette, D. (2015). *Software development metrics*. Shelter Island: Manning.

Node.js. (2011a). *How to use stream.pipe | Node.js*. Recuperado de https://nodejs.org/es/knowledge/advanced/streams/how-to-use-stream-pipe/

Node.js. (2011b). *What are streams? | Node.js*. Recuperado de https://nodejs.org/en/knowledge/advanced/streams/what-are-streams/

Parody, L. (2019, 22 noviembre). Understanding Streams in Node.js. *The NodeSource Blog #shoptalk | The Enterprise Node Company™ Providing Enterprise Node.js Training, Support, Software & Consulting, Worldwide*. Recuperado de https://nodesource.com/blog/understanding-streams-in-nodejs

Pasquali, S. (2013). *Mastering Node.js*. Birmingham: Packt Publishing. Recuperado de http://gbv.eblib.com/patron/FullRecord.aspx?p=1389335

Pressman, R. S. (2013). *Ingeniería del software: Un enfoque práctico* (Séptima edición). México, Bogotá, Buenos Aires, Caracas, Guatemala, Madrid: McGraw-Hill Education.

Pressman, R. S. y Maxim, B. R. (2015). *Software engineering: A practitioner's approach* (Eighth edition). New York NY: McGraw-Hill Education.

Rodríguez Fraga, A. (2016). *Creación y Visualización de Métricas Agiles Mediante el Uso de Herramientas Mashup.*

Schach, S. R. (2011). *Object-oriented and classical software engineering* (8th ed.). New York: McGraw-Hill.

Sommerville, I. (2016). *Software engineering* (Tenth edition). Boston: Pearson.

Sommerville, I. y Campos Olguin, V. (2011). *Ingeniería de Software* (9na. // Novena edición). México: Pearson; Pearson Educación.

Teixeira, P. (2012). *Hands-on Node.js: The Node.js introduction and API reference by Pedro Teixeira*: Leanpub.

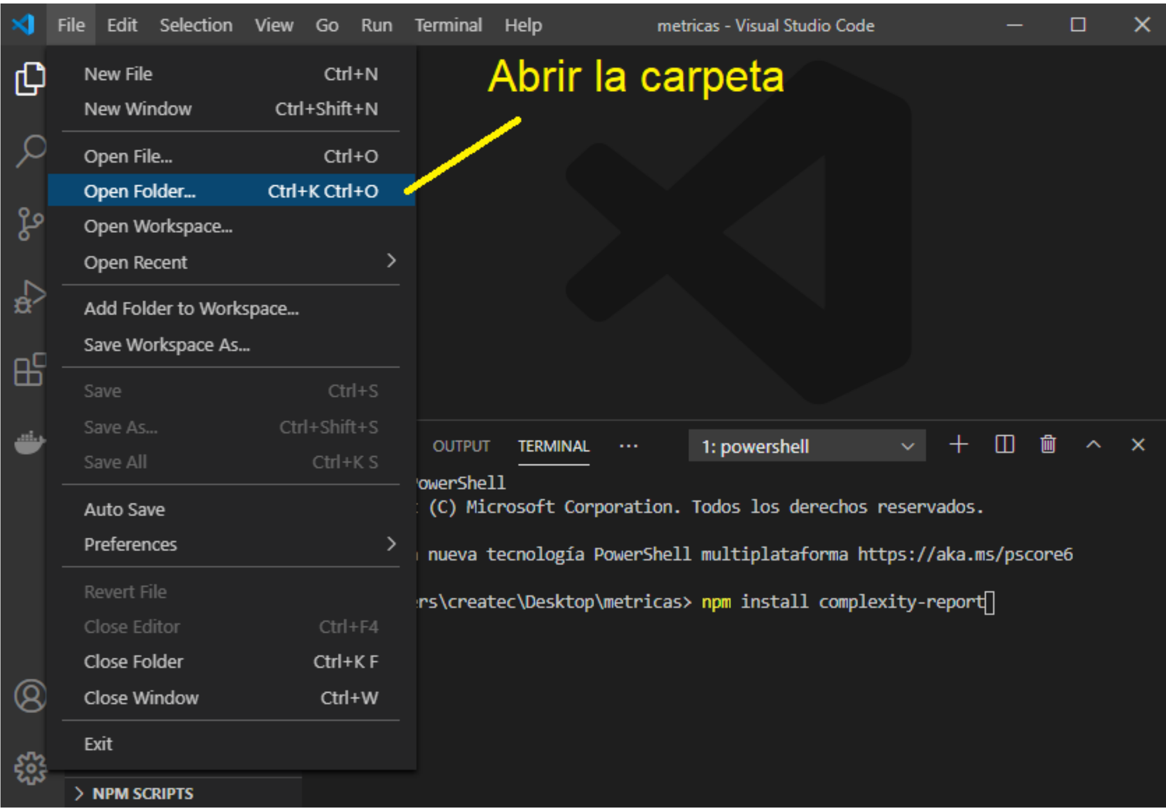
Tutorials Point. (2015). *node.js: tutorials point, simple easy learning.*

Young, A., Meck, B., Cantelon, M., Oxley, T., Harter, M., Holowaychuk, T. J. y Rajlich, N. (2017). *Node.js in action* (Second edition). Shelter Island: Manning.

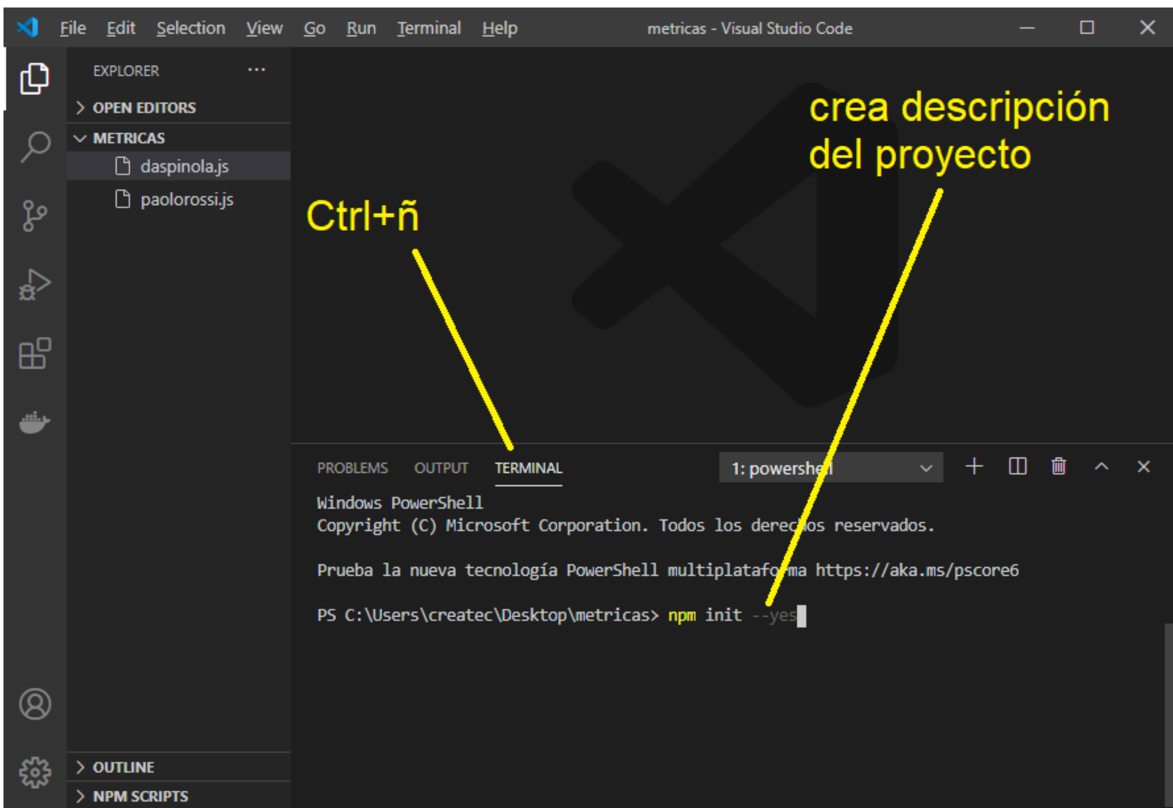
MANUAL DEL USUARIO

Una vez instalado Node.js y su entorno IDE Visual Studio Code como se indica en el manual técnico en el siguiente apartado, se procederá de la siguiente manera:

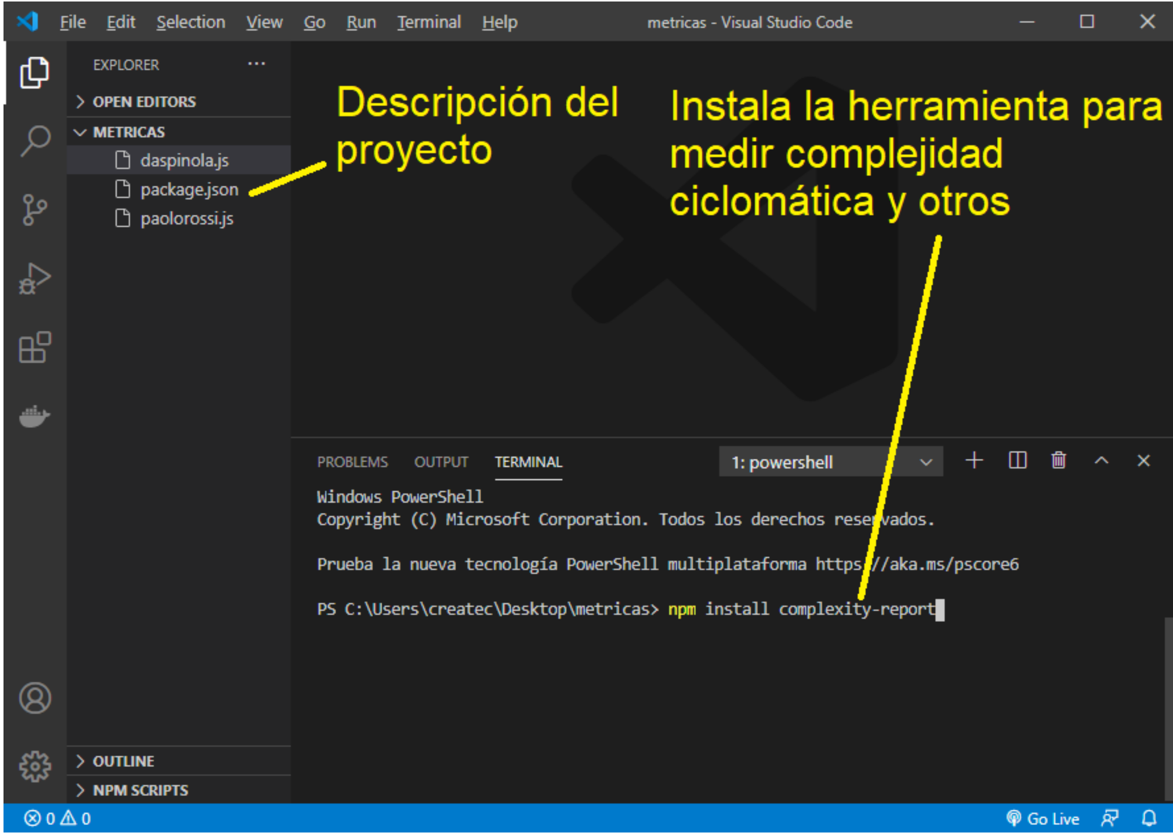
1. Crear una carpeta de trabajo e introducir los archivos que desea medir
2. Iniciar Visual Studio Code y abrir esa carpeta



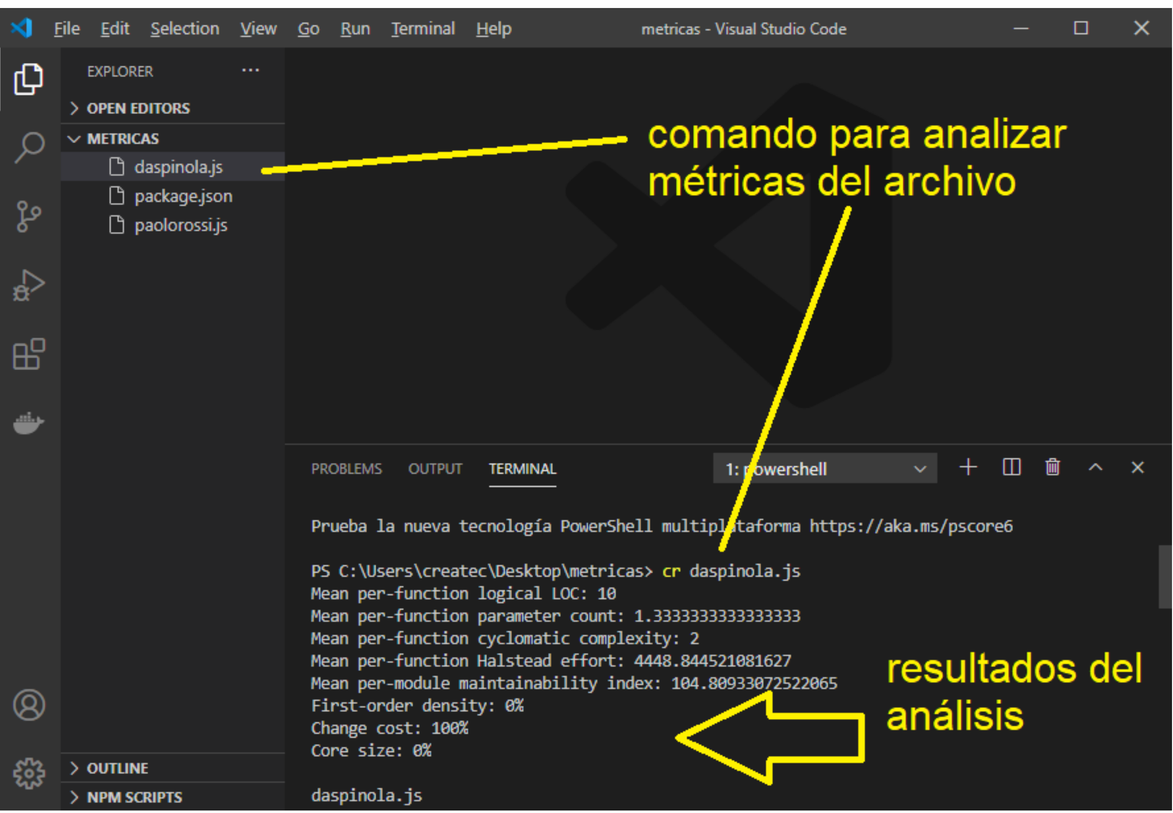
3. Abrir la terminal de línea de comandos con Ctrl + ñ en Windows, después crear el archivo de descripción del proyecto en la terminal con el comando “npm init -- y”.



4. Instalar la herramienta “complexity reports” para medir la complejidad ciclomática en línea de comandos desde la terminal



5. Analizar las métricas de un archivo:



6. Insertar el modulo nativo para el control de rendimiento como lo indica el apéndice A, poniendo atención en las sentencias subrayadas:

```
performance.mark('A');
doSomeLongRunningProcess(() => {
  performance.measure('A to Now', 'A');

  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
});
```

   Que es exactamente donde se produce el primer registro en un array entre "A hasta now" y el segundo registro entre "A hasta B", de donde obtiene las dos primeras marcas de rendimiento

MANUAL TÉCNICO

REQUERIMIENTOS DE SOFTWARE Y HARDWARE PARA NODE.JS

Para instalar JavaScript Node o Node.js será necesario cumplir requerimientos de hardware y software con los que cuenta el navegador Chrome, ya que ambos usan el entorno de ejecución V8 de Chrome.

Windows

Windows 8.1, Windows 10 o versiones superiores junto a Chrome y Node.js

Un procesador Intel Pentium 4 o posterior compatible con SSE3

Nota: Los servidores necesitan Windows Server 2008 R2, Windows Server 2012, Windows Server 2012 R2 o Windows Server 2016.

Mac

Para utilizar el navegador Chrome en Mac y Node.js, necesitas lo siguiente:

OS X Yosemite 10.10 o versiones posteriores

Linux

Para utilizar el navegador Chrome en Linux, y Node.js, necesitas lo siguiente:

Una versión de 64 bits de: Ubuntu 14.04, Debian 8, openSUSE 13.3 o Fedora Linux 24 o posteriores.

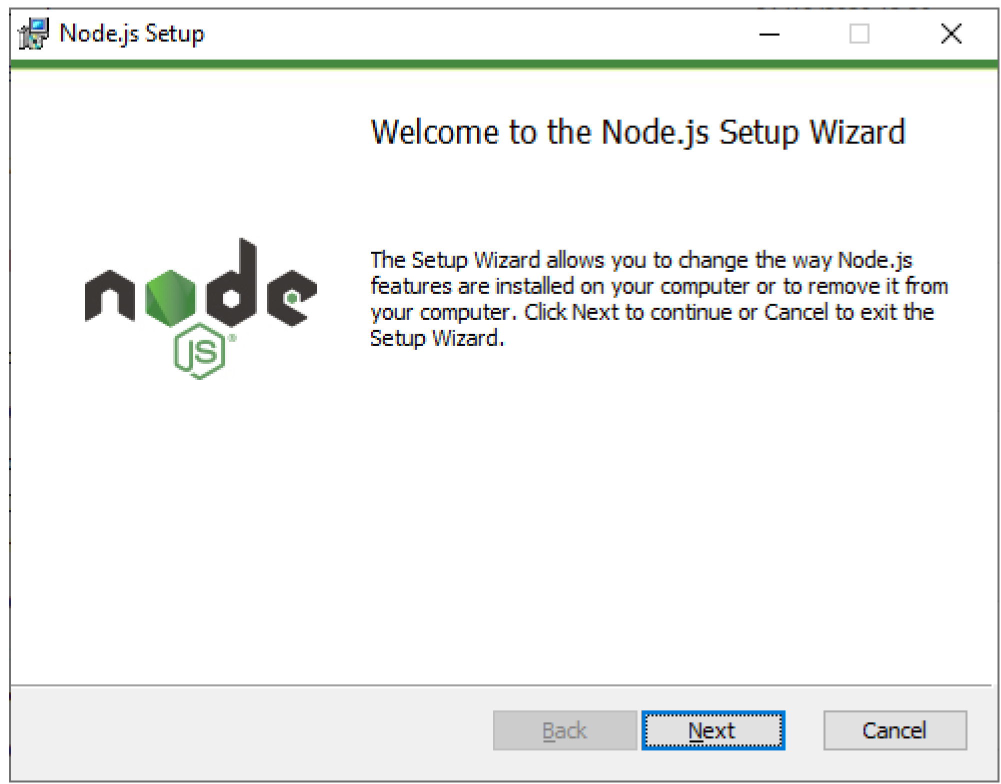
Un procesador Intel Pentium 4 o posterior compatible con SSE3

INSTALACIÓN DE NODE.JS Y VISUAL STUDIO CODE

Instalación en de Node.js y npm en Windows

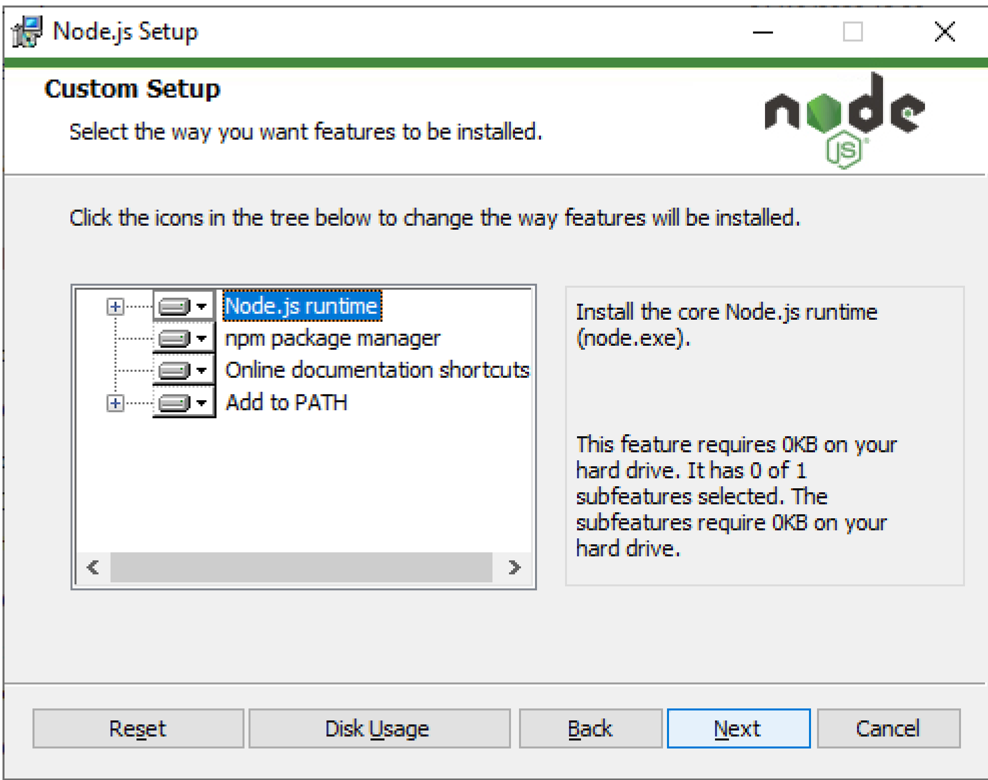
Lo primero que se debe hacer para instalar Node.js en Windows es descargar de la página oficial https://nodejs.org/en/download/ la versión que necesites.

Se descargará un archivo .msi que ejecutaremos. Este es el asistente de instalación y seguir las instrucciones recomendadas por el paquete instalador.



Durante el asistente podremos cambiar la ruta de instalación y las características que se instalarán. En cuanto a las características de la instalación, recomendamos

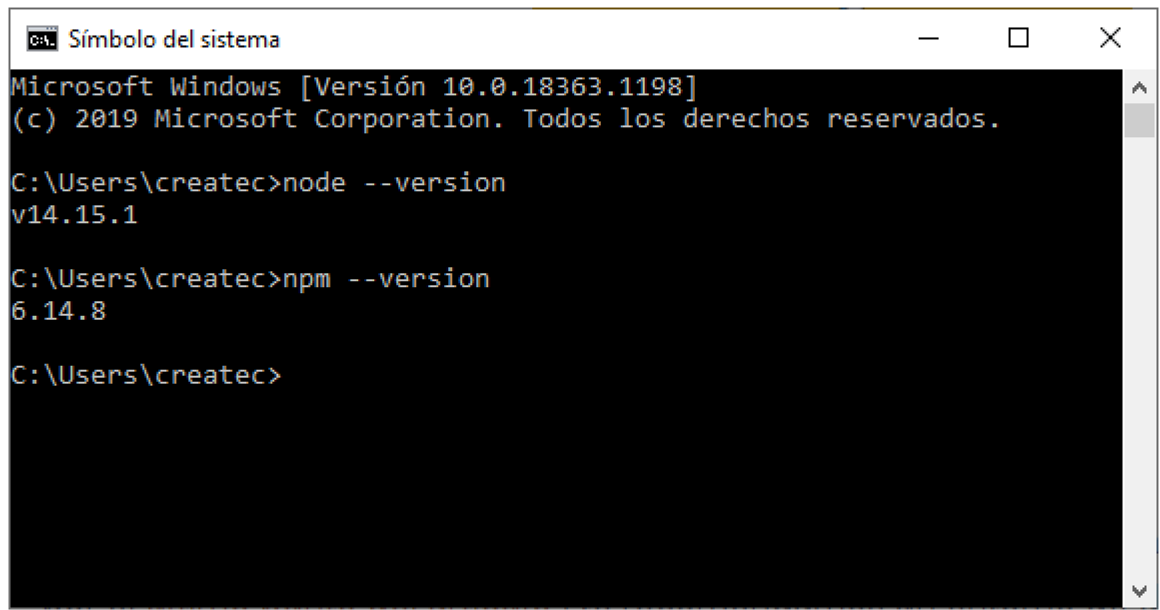
no cambiar nada dejándolo como indica por defecto, en especial el apartado npm package manager que es el gestor de paquetes de Node.js y nos será de especial utilidad a la hora de desarrollar.



Tras terminar el proceso de instalación tendremos que verificar que tenemos correctamente instalado todo lo necesario para empezar a desarrollar. Podemos comprobarlo con dos comandos:

node --version

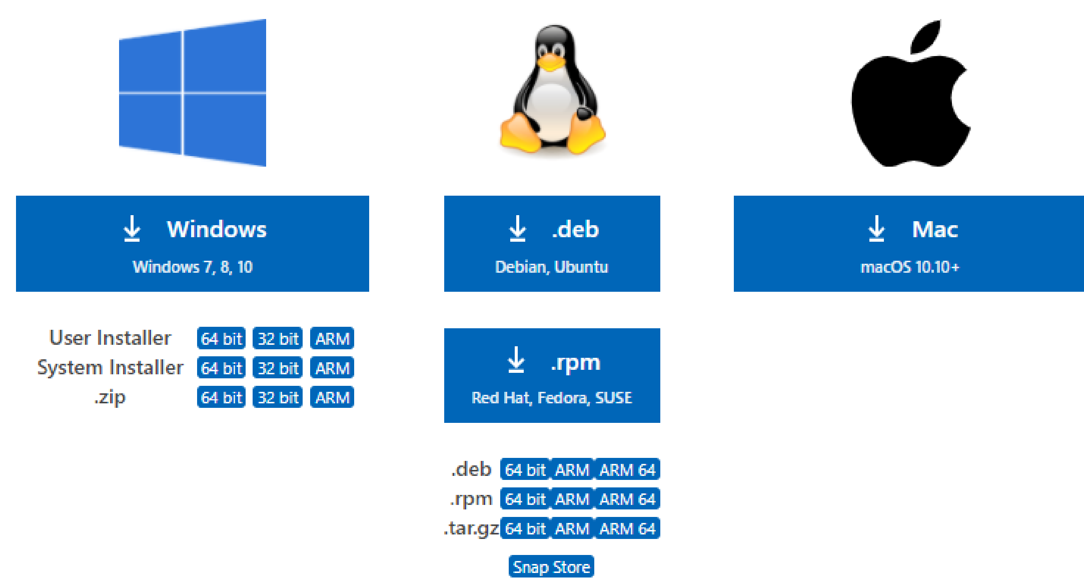
npm --version



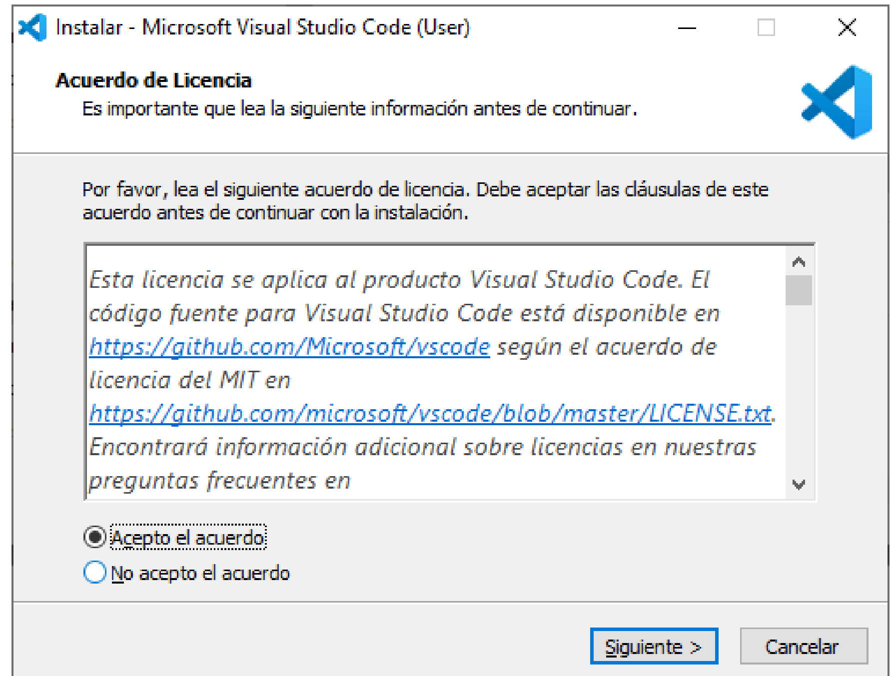
Si ejecutando esos comandos, muestra la versión correctamente es que se tiene todo listo para iniciar el desarrollo.

INSTALACIÓN DE VISUAL STUDIO CODE

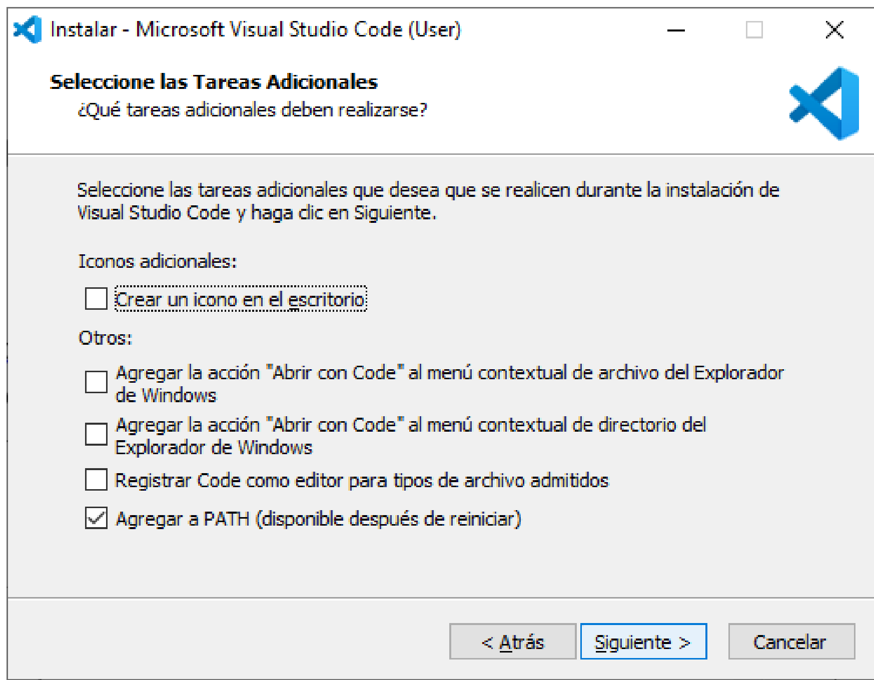
Los segundo es descargar e instalar el entorno IDE de desarrollo visual studio code, se hace la descarga del sitio https://code.visualstudio.com/.



Después de descargar ejecutar el instalador como lo hicimos anteriormente.



Asegurarse de tener marcado el último elemento que agrega la ruta de trabajo en Windows.

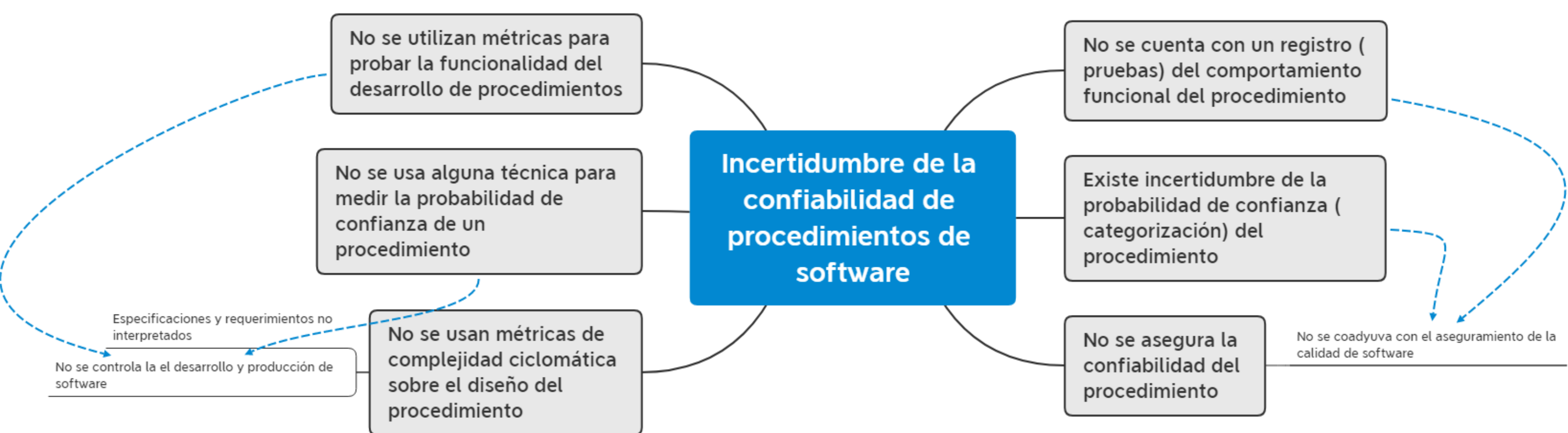


ANEXOS

ANEXOS

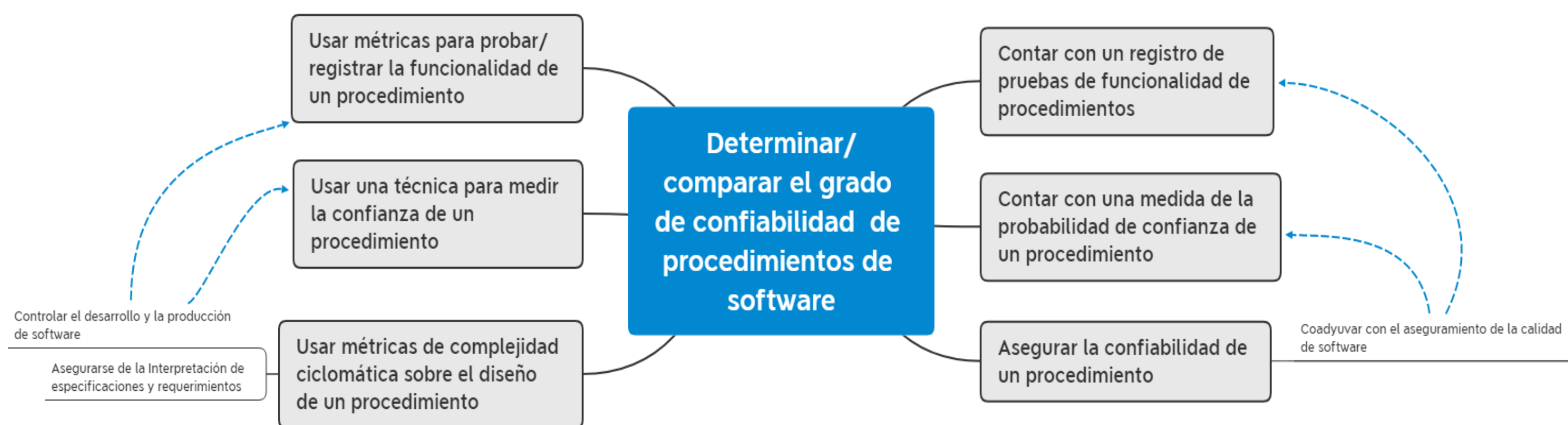
ANEXO 1

Arbol de Problemas



ANEXO 2

Arbol de Objetivos



ANEXO 3 - CÓDIGO FUENTE DEL USUARIO DE GITHUB DASPINOLA

```
const express = require('express')
const fs = require('fs')
const path = require('path')
const app = express()
app.use(express.static(path.join(__dirname, 'public')))
app.get('/', function(req, res) {
  res.sendFile(path.join(__dirname + '/index.htm'))})
app.get('/video', function(req, res) {
  const path = 'assets/sample.mp4'
  const stat = fs.statSync(path)
  const fileSize = stat.size
  const range = req.headers.range
  if (range) {
    const parts = range.replace(/bytes=/, "").split("-")
    const start = parseInt(parts[0], 10)
    const end = parts[1] ? parseInt(parts[1], 10) : fileSize-1
    if(start >= fileSize) {
      res.status(416).send('Requested range not satisfiable\n'+start+' >= '
+fileSize);
      return
    }
    const chunksize = (end-start)+1
    const file = fs.createReadStream(path, {start, end})
    const head = {
      'Content-Range': `bytes ${start}-${end}/${fileSize}`,
      'Accept-Ranges': 'bytes',
      'Content-Length': chunksize,
      'Content-Type': 'video/mp4',
    }
    res.writeHead(206, head)
    file.pipe(res)
  } else {  const head = {
      'Content-Length': fileSize,
      'Content-Type': 'video/mp4',
    }
    res.writeHead(200, head)
    fs.createReadStream(path).pipe(res)
  }
})
app.listen(3000, function () {
  console.log('Listening on port 3000!')
})
```

ANEXO 4 – CÓDIGO FUENTE DEL USUARIO PAOLOROSSI

```
/*
 * Inspired by: http://stackoverflow.com/questions/4360060/video-
streaming-with-html-5-via-node-js
 */

var http = require('http'),
    fs = require('fs'),
    util = require('util');

http.createServer(function (req, res) {
  var path = 'video.mp4';
  var stat = fs.statSync(path);
  var total = stat.size;
  if (req.headers['range']) {
    var range = req.headers.range;
    var parts = range.replace(/bytes=/, "").split("-");
    var partialstart = parts[0];
    var partialend = parts[1];
    var start = parseInt(partialstart, 10);
    var end = partialend ? parseInt(partialend, 10) : total-1;
    var chunksize = (end-start)+1;
    console.log('RANGE: ' + start + ' - ' + end + ' = ' + chunksize);
    var file = fs.createReadStream(path, {start: start, end: end});
    res.writeHead(206, { 'Content-Range': 'bytes ' + start + '-
' + end + '/' + total, 'Accept-Ranges': 'bytes', 'Content-
Length': chunksize, 'Content-Type': 'video/mp4' });
    file.pipe(res);
  } else {
    console.log('ALL: ' + total);
    res.writeHead(200, { 'Content-Length': total, 'Content-
Type': 'video/mp4' });
    fs.createReadStream(path).pipe(res);
  }
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

APÉNDICE

APÉNDICE A

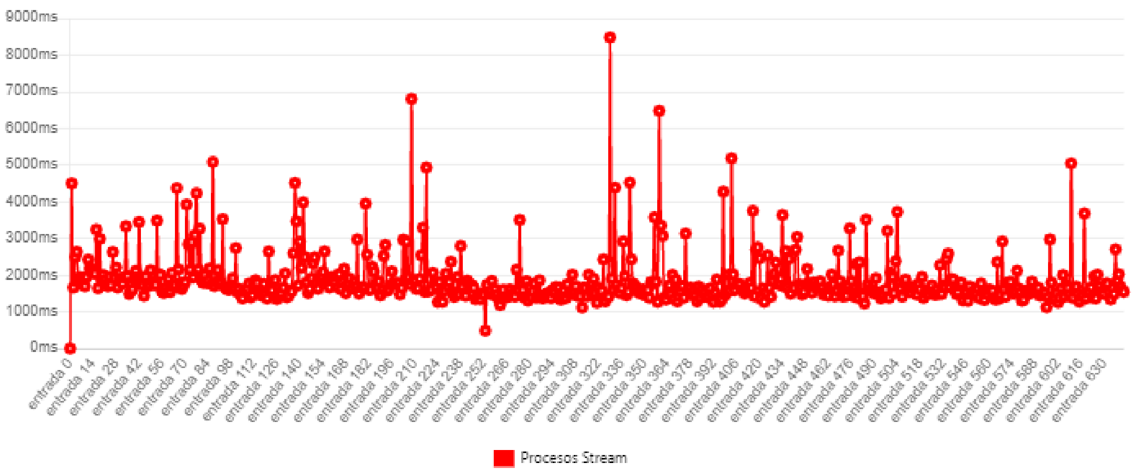
EJEMPLO DE IMPLEMENTACIÓN DEL MODULO DE RENDIMIENTO

```
const { PerformanceObserver, performance } = require('perf_hooks');

const obs = new PerformanceObserver((items) => {
  console.log(items.getEntries()[0].duration);
  performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });
performance.measure('Start to Now');

performance.mark('A');
doSomeLongRunningProcess(() => {
  performance.measure('A to Now', 'A');

  performance.mark('B');
  performance.measure('A to B', 'A', 'B');
});
```

APÉNDICE B

CAPTURA DE LA REPRESENTACIÓN GRAFICA GENERADA EN EL SERVIDOR LOCAL POR NODE.JS Y CHART.JS DEL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO PAOLOROSSI DE LA PÁGINA GITHUB.

Todas las entradas exitosas en el código del usuario: paolo rossi



APÉNDICE C

CAPTURA DE LA REPRESENTACIÓN GRAFICA GENERADA EN EL SERVIDOR LOCAL POR NODE.JS Y CHART.JS DEL RENDIMIENTO DEL CÓDIGO FUENTE DEL USUARIO DASPINOLA DE LA PÁGINA GITHUB.