UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

FACULTAD DE INGENIERÍA

"MÉTRICAS PARA EL DESARROLLO EN TIEMPO REAL DE PROGRAMAS BASADO EN AGENTES"

TESIS QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:

RITO REYES CASTILLO

SAN LUIS POTOSÍ

FEBRERO 2006

ultad de Ingeniería

Universidad Autónoma de San Luis Potosí

17 de febrero del 2005

AL LIC. RITO REYES CASTILLO
P R E S E N T E .–

En atención a su solicitud de Tema y Temario, presentado por el Dr. Héctor Gerardo Pérez González, Asesor de la Tesis que desarrollará Usted, con el objeto de obtener el Grado de **Maestría en Ingeniería de la Computación**. Me es grato comunicarle que en la Sesión de Consejo Técnico Consultivo celebrada el día 17 de febrero del presente año, fue aprobado el Tema y Temario propuesto:

TEMARIO:

"MÉTRICAS PARA EL DESARROLLO EN TIEMPO REAL DE PROGRAMAS BASADO EN AGENTES "

Agradecimientos.
Introducción.
1.- Agentes.
2.- Métricas para el desarrollo de software.
3.- Implementación del sistema centinela de métricas para el desarrollo de software.
4.- Resultados.
Conclusiones.
Anexo a. Código del Centinela de Métricas para el desarrollo de software.
Bibliografía.

"MODOS ET CUNCTARUM RERUM MENSURAS AUDEBO"

ING. JOSÉ ARNOLDO GONZÁLEZ ORTÍZ
DIRECTOR

'MOG

Av. Manuel Nava 8
Zona Universitaria
San Luis Potosí, S.L.P.
C.P. 78290
México

Índice

Agradecimientos

A DIOS por encima de todo y de todos.
Origen y destino de todo cuanto existe, fuente de vida, y esperanza del que le acepta.
Solo quien ha caminado en la oscuridad, sabe que la más pequeña luz de esperanza lo ilumina todo.

A mi padre y a mi madre, por todas las molestias, trabajos, por la entrega y dedicación en educarme.

A mis compañeros y amigos, que juntos formamos un equipo estupendo, y que con apoyo mutuo logramos lo que a solas no podríamos, la suma es mayor que las partes.

A mis maestros por su dedicación y entrega, lo que me ha permitido crecer profesionalmente.

A toda persona que en algún momento de la vida a compartido el caminar, y que a través de ese contacto efímero se han modificado un poco nuestras vidas, y en cada contacto nos hemos influenciado, solo te deseo lo mejor.

Tantos siglos, tantos mundos, tanto espacio,...
y coincidir.

1. Introducción

Antes de que la máquina computadora fuese una realidad ya existían algunos algoritmos mediante los cuales se realizaban algunas operaciones matemáticas, como ejemplo podemos enumerar "La Criba de Eratóstenes", que consiste de un proceso de eliminación para encontrar números primos, éste es uno de los algoritmos más antiguos que se conocen.

Sin embargo los nuevos algoritmos basados en el uso de una máquina computadora son significativamente más complicados, hoy día existe un incremento constante de nuevos dispositivos periféricos y de los mismos procesadores, además los lenguajes cambian e incluyen instrucciones complejas con grandes necesidades de proceso, las comunicaciones en empresas con departamentos separados físicamente representan necesidades que deben contemplarse, por último se añade la constante necesidad de actualización en equipos y sistemas, un proceso que en una empresa moderna debe realizarse al menos cada tres años.

Todo lo anterior crea una complejidad creciente tanto en el software, como en el proceso de desarrollo del mismo, esta complejidad involucra principalmente a los programadores que tradicionalmente son los encargados de escribir el software en un lenguaje específico, éstos desarrollan programas que normalmente deben incluir la documentación suficiente para que cualquier otro programador pueda entender y así efectuar correcciones al mismo, pero no siempre ha sido así, en los años 60's los programadores no incluían documentación adecuada, en muchos casos no incluía ninguna, esta deficiencia y el incremento de la complejidad de los sistemas originó la llamada "crisis del software", en ese entonces en una conferencia en 1968 en EUA se creó la noción de la hoy llamada Ingeniería de Software [SOMMER].

En este documento se propone una disciplina de desarrollo de software que

implica la toma de métricas durante dicho desarrollo, con algunas de las métricas indicadas por Ian Sommerville [SOMMER], y con un registro del avance durante el desarrollo, lo que llevará a conocer la productividad de cada persona o equipo de desarrollo, y así obtener elementos para la toma de decisiones con un mayor grado de exactitud.

En el capitulo 1 se verán los principios básicos de la ingeniería de software, cómo puede auxiliar ésta en el desarrollo de sistemas, en que situación se encuentra la ingeniería de software y las propuestas de este documento.

En el capitulo 2 se desarrolla el tema de agentes, las métricas de software, que métricas son las más utilizadas en la industria de software, entre otros, como base para el desarrollo, en el capitulo 3, de un sistema para auxiliar de toma de métricas, estableciendo ciertas métricas para ello.

En el capitulo 4 se tienen los resultados del sistema medido con el mencionado programa de toma de métricas, y algunas observaciones de éstos resultados.

Posteriormente en el capitulo 5 se muestran las conclusiones del presente trabajo, además de mencionar que posibles trabajos a futuro pueden esperarse para el sistema de toma de métricas, para ampliar la versatilidad del mismo.

En el capitulo 6 se tiene un anexo con las clases relevantes (en disco), una descripción del sistema de toma de métricas, además del manual de usuario del mismo sistema.

1.1 Conceptos de ingeniería de software

Software

El software es un conjunto de elementos necesarios para que un sistema funcione correctamente [SOMMER], éstos son:

- El programa o conjunto de programas.
- Los archivos de configuración necesarios para la correcta ejecución del sistema.
- El sistema de documentación que describa la estructura del proyecto.
- La documentación necesaria para que el usuario utilice el sistema.
- Actualmente se incluyen sitios web para que el usuario descargue actualizaciones, configuraciones, documentación, etc.

Ingeniería

La ingeniería debe su nombre al ingenio, a la forma práctica de hacer algo, a lo largo de la historia de la humanidad, el ingenio ha acompañado al desarrollo de la misma, encontrando maneras de ejecutar una función para el problema específico y posteriormente refinando el mismo proceso para ahorrar tiempo, dinero o esfuerzo, o para obtener mayor calidad.

La ingeniería hace que las cosas funcionen, aplicando métodos, teorías y herramientas de forma selectiva, trabajando con restricciones financieras y organizacionales, no importa el área de la ingeniería de que se trate, cada una de ellas se rige de la misma forma.

Ingeniería de software

La Ingeniería de Software es la disciplina que comprende todos los procesos técnicos del desarrollo de software, la administración de proyectos, el desarrollo de herramientas, métodos y teorías de apoyo para la producción de software, desde las etapas iniciales de la especificación del sistema, hasta el mantenimiento

de éste después de que se entrega, es decir cuando esta siendo utilizado por el cliente.

Un ingeniero de software adopta un enfoque sistemático y organizado en su trabajo, ya que esta es la forma más efectiva para producir software de alta calidad y de que esta calidad sea repetible.

La ingeniería de software es el establecimiento y uso de principios robustos de la ingeniería con el fin de obtener económicamente software que sea fiable y que funcione eficientemente sobre máquinas reales [PRESS].

1.2 Estado del arte en métricas de Ing. de Software

La toma de métricas otorga parámetros para la toma de decisiones administrativas y para mejorar los procesos de desarrollo de software [AIS].

La ingeniería de software marca una serie de elementos que deben ser cubiertos para que pueda existir un sistema de software de calidad, como se menciona en este apartado, estos elementos abarcan todo el ámbito de desarrollo de un producto de software.

1.2.1 Requerimientos

Se debe consultar al cliente para especificar sus necesidades, analizarlas, documentarlas, verificarlas y establecer las restricciones, el cliente no siempre sabrá especificar lo que necesita, para esto el ingeniero de software tiene una o varias entrevistas con el cliente, con los usuarios finales del sistema, y con cada persona involucrada en el mismo, para así ayudarles a definir especificamente cada una de sus necesidades de manera correcta, y así evitar posibles problemas en el diseño e implementación del sistema.

1.2.2 Diseño

Una vez especificados los requerimientos, la fase de diseño analiza la información para establecer de que manera se va a diseñar el sistema, entre algunos aspectos están: El desempeño, la seguridad, la protección, la disponibilidad, la mantenibilidad. Una vez que se seleccionan que aspectos son importantes se puede establecer que tipo de diseño se va a utilizar.

1.2.3 Verificación y validación

Se les llama así a los procesos de comprobación y análisis que aseguran que el software esté acorde con su especificación y cumpla las necesidades del cliente, éstas comprenden todo el ciclo de vida de un sistema.

La verificación consiste en comprobar que el software cumpla con las especificaciones.

La validación consiste en asegurar que el producto cumpla con las expectativas del cliente.

Dentro de este proceso se utilizan dos técnicas de comprobación y análisis de sistemas:

- Las inspecciones del software: Analizan y comprueban las representaciones del sistema como el documento de requerimientos, los diagramas de diseño y el código fuente del programa. Se llevan a cabo pruebas sistemáticas de los programas, lo que significa un proceso largo y caro, en esta verificación inicial se pueden encontrar muchos de los errores del sistema.
- Las pruebas del software: Comprenden el llevar a cabo una implementación del software con datos de prueba y examinar las salidas del software y su comportamiento operacional para comprobar que se desempeñe conforme a lo requerido.

A su vez existen dos tipos diferentes de pruebas que se utilizan en varias etapas del proceso del software:

- Las pruebas de defectos: Pretenden encontrar las inconsistencias entre un programa y su especificación.
- Las pruebas estadísticas: Se utilizan para probar el desempeño y la fiabilidad del programa y comprobar cómo trabaja bajo condiciones operacionales. Las pruebas se diseñan para reflejar las entradas de los usuarios y su frecuencia.

La meta fundamental del proceso de verificación y validación es generar la confianza de que el sistema se ajusta a los propósitos, aunque esto no significa que el sistema este libre de errores, sino que el sistema es suficientemente confiable para los fines que se pretende.

Aclarando lo anterior, el nivel de confianza requerido depende de qué tan importante es el software para una organización, aunque tradicionalmente los usuarios aceptan cierta cantidad de errores por que el sistema les beneficia aún con los errores mencionados, en la actualidad no se acepta entregar un sistema con un nivel alto de errores.

La existencia de defectos se establece a través de la verificación y la validación pero se tiene que depurar el sistema para asi localizar y corregir estos defectos.

Una forma de verificar y/o validar el software son las métricas de software, que pueden ser de producto y de proceso, la diferencia entre éstas es que las métricas del proceso se obtienen durante el desarrollo del proyecto, es decir, cuando los programadores están codificando el proyecto, y las métricas del producto se obtienen del producto terminado, más adelante se hablará de las métricas de producto, pero tomadas durante el proceso.

1.2.4 Administrador de software

Siendo el administrador de software el encargado de establecer parámetros para el desarrollo dentro de una empresa de desarrollo de software, es necesario que tenga elementos para la toma de decisiones administrativas, entre éstos, el personal y sus capacidades son relevantes ya que el activo más grande de una empresa de software es el personal, este representa el capital intelectual y el administrador de software debe asegurar que la organización tenga los mejores beneficios posibles al invertir en personas.

Ésto se cumple cuando las personas son respetadas en una organización, tienen un nivel de responsabilidad y se les asignan premios de acuerdo con sus capacidades.

Todo ser humano tiene motivaciones individuales pero también existen motivaciones de grupo, familia, cultura, etc., un administrador debe tener en cuenta cada una de ellas para así poder obtener el máximo rendimiento del personal, sin llegar a la explotación ya que si se presiona demasiado al personal este bajará su rendimiento, por ejemplo, solicitar horas extras siempre redunda en una tensión, esta crece hasta que el personal baja su rendimiento, por cansancio, por falta de recreación, y puede empezar a faltar como mínimo o hasta renunciar como resultado de esa tensión.

La selección del personal para un proyecto, normalmente depende del presupuesto, del tiempo y de la disponibilidad, esto limita el uso de personal con amplia experiencia y habilidades, pero aún así se tiene que realizar, algunos factores para la selección de personal son: experiencia en el dominio de la aplicación, experiencia en la plataforma, experiencia en el lenguaje de programación, soporte educativo, habilidad de comunicación, adaptabilidad, actitud y personalidad (compatibilidad).

El tamaño de la oficina, el mobiliario, el equipo, la temperatura, la humedad, la brillantez y calidad de la luz, el ruido, el grado de privacidad son algunos elementos que afectan el comportamiento del personal en una empresa, la arquitectura del edificio y el tipo de organización afectan a las comunicaciones.

La importancia de lograr un entorno de trabajo adecuado, redunda en la permanencia del personal, es decir, si el personal rota continuamente, el costo de reclutar y capacitar se incrementa.

Para lograr un entorno adecuado algunos de los factores más importantes son [SOMMER]:

- Privacidad: los programadores requieren de un área donde se puedan concentrar y trabajar sin interrupción.
- Conciencia del exterior: Las personas prefieren trabajar con luz normal y con una vista del entorno exterior.
- Personalización: Los individuos adoptan diferentes prácticas en el trabajo y tienen diferentes opiniones sobre la decoración, es necesario permitirle personalizar el lugar de trabajo.

Dependiendo del tipo de estructura de la empresa se pueden establecer espacios comunes y áreas de trabajo, siempre teniendo en cuenta la comunicación y que el personal se sienta en un entorno agradable.

1.2.5 El modelo de madurez de la capacidad del personal

El nivel de experiencia del personal y su supervisión adecuada por parte del administrador y del supervisor de software, tienen que ver directamente con la capacidad de producir un producto de software de calidad repetible.

El elemento fundamental en todos los proyectos de software es el personal, independientemente del organigrama y de las técnicas de coordinación y

comunicación [PRESS].

El activo más importante en una empresa de software, es el personal, que sea adecuadamente capacitado y organizado de manera eficiente provocará un proceso y un producto de calidad [AIS].

El modelo de madurez de la capacidad del personal es un marco de trabajo para mejorar la forma en que una organización administra sus activos humanos, Ian Sommerville [SOMMER] propone cinco niveles que se enuncian a continuación:

a) ***Inicial***: Se utilizan prácticas informales de administración de personal.

- Se infunden disciplinas básicas en las actividades de trabajo.

b) ***Repetible***: Se establecen políticas para del desarrollo de la capacidad del personal (Compensación, Capacitación, Administración del desempeño, Aprovisionamiento de personal, Comunicación, Ambiente de trabajo).

-Se identifican competencias primarias y se alinea la fuerza de trabajo con ellas.

c) ***Definido***: Estandarización de las mejores prácticas de administración de personal a lo largo de la organización (cultura participativa, prácticas basadas en la competencia, desarrollo profesional, desarrollo de competencias, planeación de la fuerza de trabajo, análisis del conocimiento y las habilidades).

- Se administra cuantitativamente el crecimiento organizacional en cuanto a las aptitudes de la gente y se establecen equipos basados en la competencia.

d) ***Administrado***: Se establecen e introducen las metas cuantitativas para la administración de personal (alineación del desempeño organizacional, administración de la competencia organizacional, prácticas basadas en el equipo, formación de equipos, asesoramiento).

- Se introducen métodos de mejora continua para el desarrollo del personal y la competencia organizacional.

e) ***Optimizado***: Existe un enfoque continuo en la mejora de la competencia individual y la motivación de la fuerza de trabajo (innovación continua de la fuerza de trabajo, asesoramiento, desarrollo de competencia personal).

Los objetivos del modelo de madurez de la capacidad del personal son:

- Mejorar la capacidad de las organizaciones de software incrementando la capacidad de su fuerza de trabajo.
- Asegurar que la capacidad de desarrollo de software sea un atributo de las organizaciones más que de pocos individuos.
- Alinear las motivaciones de los individuos con las de la organización.
- Retener los activos humanos dentro de la organización.

La medición continua da parámetros para conocer el desarrollo de una empresa, lo que es una manera excelente de autoevaluación, si se desea calidad en software se debe incluir el factor humano y con ello el desarrollo personal continuo.

1.2.6 Estimación de costos del software

El administrador debe proyectar el costo de un proyecto sobre la base de estimaciones y su experiencia, para realizar ésta acción debe colectar toda la información posible, las métricas de software son de importancia ya que proveen información acerca de la productividad del personal, pero esto no es todo lo que debe considerar, sino también: ¿Cuánto cuesta un proyecto de software?, ¿Cómo

se puede calcular el esfuerzo?, ¿En cuanto tiempo se termina el proyecto?. Es relativamente fácil calcular los costos de iluminación, renta, compra, mantenimiento, viajes, alimentación, instalación, recursos (bibliotecas, consultas, etc.), seguro social, pensiones, electricidad, equipo, software, etcétera.

Para la mayoria de los proyectos de software el costo dominante es el esfuerzo (los costos de pago a los ingenieros de software), de aqui que calcular los costos de software se debe llevar a cabo de forma objetiva con el fin de predecir lo más preciso posible cuánto le costará al contratista desarrollar el software y es responsabilidad del administrador principal de la organización y de los administradores del proyecto de software [SOMMER].

- *Productividad*: Para poder valorar si los procesos o mejoras tecnológicas son efectivas es necesario poder estimar la productividad en el proceso de desarrollo de software, generalmente estas estimaciones se basan en medir algunos atributos del software y dividir el resultado entre el esfuerzo total requerido para su desarrollo.

Existen dos tipos de medidas utilizadas:

a) Medidas orientadas al tamaño:
   Numero de lineas de código, número de instrucciones de código objeto o el número de páginas de la documentación del sistema.

b) Medidas relacionadas con la función:
   Puntos de función y puntos de objeto.

En cuanto a medidas por tamaño se pueden medir las lineas de código del proyecto completo y dividirlo entre el tiempo para tener un estimado de productividad, horas-hombre por ejemplo, pero algunas técnicas descuentan las lineas vacias, o las de comentarios, otras solo cuentan las lineas que contienen una instrucción ejecutable o de datos, medir las lineas de diferentes

lenguajes es engañoso, es necesario utilizar solo una técnica para el proyecto completo y/o para cada lenguaje.

- *Técnicas de estimación.-* No existe una forma simple de efectuar una estimación precisa del esfuerzo requerido para desarrollar un sistema de software, pero si existe una dificultad fundamental para valorar la precisión de cada uno, ésta dificultad es que por ser una estimación se pretende conocer lo que va a suceder en base a lo ya ha pasado, es decir, dar un pronostico en base a la experiencia previa, que sigue siendo solo eso, un pronostico y por tanto no un hecho.

  Las técnicas de estimación de costos son [SOMMER]:

  a) Modelado del algoritmo de costos

     Se desarrolla un modelo utilizando información histórica de costos que relaciona alguna métrica de software con el costo del proyecto.

  b) Opinión de expertos

     Se consultan varios expertos en las técnicas de desarrollo de software propuestas y en el dominio de la aplicación.

  c) Estimación por analogia

     Esta técnica es aplicable cuando otros proyectos del mismo dominio de aplicación se han completado.

  d) Ley de Parkinson

     Establece que el trabajo se extiende para llenar el tiempo disponible, el costo se determina por los recursos disponibles más que por los objetivos logrados.

  e) Asignar precios para ganar

     El costo se estima independientemente de lo que el cliente esté dispuesto a pagar por el proyecto.

Cada técnica se puede abordar utilizando un enfoque descendente o ascendente:

- El descendente inicia a nivel de sistema examinando la funcionalidad total del producto y como la funcionalidad se propaga al interactuar con las subfunciones.
- El ascendente inicia al nivel de componente sumando el esfuerzo requerido para cada uno de estos, al final se suman todos los esfuerzos para generar el total requerido para el desarrollo del sistema.

La desventaja del enfoque descendente es que subestima los costos de resolver problemas técnicos difíciles asociados con componentes específicos como las interfaces para hardware, no existe justificación detallada de la estimación producida. Por otro lado el enfoque ascendente produce la justificación mencionada pero tiende a subestimar los costos de las actividades del sistema como la integración, además es más costosa y debe existir un diseño inicial para identificar cada componente.

Si bien cada técnica tiene ventajas y desventajas, en un proyecto grande se deben utilizar varias técnicas de estimación y comparar los resultados, si estos son muy diferentes, se presume entonces de que no se tiene suficiente información y se debe buscar más datos y repetir el proceso de estimación.

Las métricas de software generan parámetros para obtener estimaciones de esfuerzo (líneas de código, complejidad ciclomática, entre otros), a partir de esta información, además de su experiencia, el administrador y el equipo de desarrollo pueden trazar un calendario estimado sobre bases más reales.

1.2.7 Administración de la calidad

Una empresa profesional requiere de niveles altos de calidad y requiere también que esa calidad sea un estándar, sin embargo la calidad en software no es algo simple de definir.

Calidad es que el producto desarrollado cumpla con sus especificaciones, pero en software la especificación de un producto depende directamente de lo que el cliente requiere, pero la empresa de software aumenta esos requerimientos con los propios, la calidad no se puede especificar de forma exacta, aunque un producto esté acorde a los requerimientos, el usuario no reconoce el producto como de alta calidad [SOMMER].

Un buen administrador desarrolla una cultura de calidad, donde cada persona encargada de un área del producto es motivada a responsabilizarse de la calidad de su trabajo y de desarrollar nuevos enfoques de mejora de la calidad, esto crea un producto de alta calidad, sin embargo un administrador sabe que algunos aspectos como la elegancia y transparencia no están incluidos en los estándares de calidad ni se pueden medir directamente.

1.2.7.1 Aseguramiento y estándares de calidad

Si se desea la calidad como una cultura de trabajo, se debe asegurar que la calidad exista siempre, para esto hay dos tipos de estándares que se establecen como parte del proceso de aseguramiento de la calidad [SOMMER]:

- Estándares del producto: incluye estándares de documentos, documentación y codificación.
- Estándares del proceso: incluye definiciones de los procesos de especificación, diseño, validación y descripción de documentos a generar.

Los estándares de software son importantes por que proveen un conjunto compacto de prácticas que evitan la repetición de errores, capturan el conocimiento de valor para la organización, proveen un marco de trabajo, capturan las mejores prácticas, etc. de esta manera el control de calidad es una forma de mantener los estándares en cada paso del desarrollo.

Todos estos estándares pueden considerarse como burocráticos e irrelevantes para el personal creativo, el llenar formularios tediosos, el mantener bitácoras de

trabajo no es labor "apropiada" para un desarrollador, un administrador conoce esta situación y para evitarla se debe involucrar al ingeniero de software en el desarrollo de estándares para que se comprometan en el mismo y tengan la motivación de seguirlo, además el administrador debe revisar y modificar los estándares de forma regular para reflejar las nuevas tecnologías, un manual de estándares, continuamente mejorado es esencial, proveer de herramientas de software para apoyar los estándares, también es importante [SOMMER].

1.2.7.2 Estándares de documentación

Éstos tienen una importancia relevante ya que son la única forma tangible de representar al software y al proceso del software.

Estándares del proceso de documentación, estándares del documento, y estándares de intercambio de documentos

1.2.7.3 Planeación de la calidad

Desde el inicio del proyecto se define la calidad deseada del producto, se seleccionan los estándares organizacionales apropiados y el proceso de desarrollo.

La estructura del plan de calidad es [SOMMER]:

- Introducción del producto: Contiene la descripción del producto, el mercado al que va dirigido y las experiencias de calidad.
- Planes del producto: Contiene las fechas de terminación del producto y responsabilidades importantes junto con los planes para distribución y servicio.
- Descripciones del proceso: Contiene los procesos de desarrollo y de servicio a utilizar para el desarrollo y administración del producto.
- Metas de calidad: Contiene las metas y planes de calidad para el producto, incluye una identificación y justificación de los atributos de calidad importantes del mismo.
- Riesgos y administración de riesgos: Contiene los riesgos clave que podrían afectar la calidad del producto y las acciones para abordar estos riesgos (plan de contingencia).

Seguridad	Comprensión	Portabilidad
Protección	Experimentación	Usabilidad
Fiabilidad	Adaptabilidad	Reutilización
Flexibilidad	Modularidad	Eficiencia
Robustez	Complejidad	Aprendizaje

Fig. 1.2.7.3.1. Atributos de calidad del software [SOMMER].

No es posible optimizar todos los atributos de un sistema (ver Fig. 1.2.7.3.1), pero si es necesario elegir los adecuados al producto y expresarlos en el plan de calidad, dependiendo del mismo proyecto, qué es lo primordial y qué es lo secundario, además de los estándares de la empresa [SOMMER].

1.2.7.4 Control de la calidad

Para lograr una calidad en todo el proceso de desarrollo, se tiene que supervisar cada parte del mismo, esto implica revisiones de calidad, donde el software, la documentación y los procesos, son medidos y revisados por un grupo de personas, comprobando el seguimiento de los estándares, tomando nota de las desviaciones y poniendo estas a consideración del administrador del proyecto.

1.2.7.5 Medición de software

Para medir el software es necesario asignar un valor numérico a algún atributo de un producto de software (número de lineas de código) o a un proceso de software (puntos de función); y, al comparar este valor estadisticamente (del mismo producto de software o de otros productos) y con los estándares de la empresa se pueden obtener conclusiones acerca de la calidad del software o del proceso.

Las anteriores son llamadas métricas de producto, que se refiere a obtener valores del código fuente del producto terminado.

A su vez éstas pueden ser [SOMMER]:

- Dinámicas (mediciones del programa en ejecución), dentro de este ámbito están: El tiempo de corrida de un proceso, Tiempo de inicio del sistema, número y tipo de caidas del sistema, tiempo de acceso a datos, tiempo de respuesta , etc.
- Estáticas (mediciones hechas al diseño, al código del programa o la documentación), dentro de éste ámbito están: Tiempo estimado de desarrollo, esfuerzo y tiempo usado para reparación de fallos, longitud de código, complejidad ciclomática, longitud y claridad de la documentación, etc.

Por otra parte existen también las métricas del proceso, que se refiere a tomar datos cuantitativos de los procesos del software, esto para la mejora continua del proceso de desarrollo, datos como el tiempo y los recursos dedicados a un proceso en particular, o el número de ocurrencias de un evento, todos enfocados a responder preguntas y confirmar si las mejoras del proceso ayudan a cumplir la meta deseada [SOMMER].

Cabe mencionar que el obtener métricas no es un proceso normal hoy en dia, pocas compañías llevan a cabo esta técnica, por un lado no hay un consenso de los beneficios obtenidos y por otro no existen muchas herramientas para la recolección y el análisis de datos [AIS].

La única forma racional de mejorar cualquier proceso es medir atributos del proceso, desarrollar un conjunto de métricas significativas según estos atributos y entonces utilizar las métricas para proporcionar indicadores que conducirán a una estrategia de mejora [PRESS].

1.3 Propuesta y objetivos

Los objetivos de este trabajo son:

1. Generar un sistema basado en agentes de toma de métricas para el desarrollo de programas, como auxiliar para los administradores, supervisores o desarrolladores.
2. Fomentar una cultura de trabajo para el desarrollo programas por medio del uso de métricas.
3. Crear herramientas para procesos de mejora continua.
4. Otorgar parámetros al personal de desarrollo para conocer su propia productividad.
5. Generar elementos para la toma de decisiones del administrador del proyecto o del administrador general de una empresa de software.

Se propone un sistema basado en agentes, que toma algunas métricas relevantes de los archivos fuente de programas hechos en lenguaje Java (JBuilder EE, Ver. 3), verificando continuamente el desarrollo de un proyecto y registrando las métricas obtenidas cuando ocurra un cambio en el mismo, y a petición del usuario mostrar una estadistica de las métricas tomadas por proyecto.

Este programa se propone como herramienta auxiliar en el desarrollo y administración de sistemas de software, de esta manera se puede conocer la productividad de un programador o un grupo de programadores y de ahi la toma de decisiones administrativas serán más fáciles, prácticas, reales y de cierta forma tangibles.

2. Agentes y métricas para el desarrollo de software

2.1 Agentes

Agente es el que realiza una acción. El que actúa en representación y a beneficio de otro (agente artístico, comercial, inmobiliario, de seguros, de bolsa, etc).

Un agente actúa en beneficio de su representado y sus acciones e interacciones en el entorno van encaminadas a otorgar un beneficio específico y constante [ATSA].

Para definir más un agente, observaremos a un agente de viajes y a un agente de bienes raíces ya que ambos actúan en beneficio de otros, en el caso de agente de viajes, el representado es el hotel o la compañía de aviación, que ofrecen servicios y éstos son promocionados por el agente, encontrando viajeros que desean utilizar el servicio de los primeros, el agente de raíces actúa en beneficio del que desea vender su casa ofreciendo asesoramiento en cuanto al valor, los tramites y encontrando al comprador con capacidad de pagar.

Además de actuar en beneficio de otros, un agente también tiene autonomía, volviendo a los ejemplos, un agente de bienes raices puede investigar las caracteristicas de una casa desocupada para encontrar un posible cliente, un agente de viajes puede contactar diferentes cadenas de hoteles y empresas de aviación con las que no tiene contacto y llegar a un arreglo para ampliar su oferta de viajes.

Otra caracteristica es la proactividad o reactividad, significa qué tanto un agente actúa en el medio y qué tanto responde al medio, un agente de raices puede ofrecer una casa simplemente poniendo un letrero en el patio de la casa en venta o puede anunciar en el periódico local, estatal, o nacional, incluso en televisión o en Internet, si solo hace un letrero al agente esperará mucho tiempo y será reactivo al medio, pero si hace todo lo demás el agente será más proactivo, de

igual manera un agente de viajes puede promocionar sus paquetes o puede esperar a que al cliente llegue a su oficina a preguntar.

Un agente puede tener otras caracteristicas como aprendizaje, cooperación y movilidad, esto significa que el agente puede modificar su comportamiento de acuerdo con la experiencia obtenida y/o almacenada en algún lugar, o puede cooperar con otros agentes para una meta común y más grande, además de que puede moverse fisicamente hacia otras computadoras en búsqueda de información [NAVA].

En resumen un agente es una entidad computacional que actúa en beneficio de otras entidades con un cierto comportamiento autónomo, que realiza su actividad con un grado de reactividad, proactividad o ambas, y que puede exhibir o no un cierto nivel de aprendizaje, cooperación o movilidad [WEISS].

2.1.1 Agente de software

Es un sistema de software que está situado en cierto entorno y que tiene capacidad de actuar de manera autónoma en ese entorno para satisfacer los objetivos de su diseño. El agente detecta el entorno a través de sensores y dispone de un repertorio de acciones que puede utilizar de acuerdo a sus objetivos y puede modificar el entorno. Tiene sus orígenes en la inteligencia artificial, los demonios, y los sistemas expertos [NWANA].

Los Agentes se clasifican en sistemas llamados [NWANA]:
Colaborativos, de Interfaz, Móviles, de Información, Reactivos, Hibridos e Inteligentes.

Para fines de este proyecto se seleccionó el agente reactivo ya que es el que más se acerca en definición y en comportamiento a las necesidades del mismo.

2.1.2 Agentes Reactivos

Reactividad significa capacidad de mantener una interacción continua con el ambiente, y respondiendo en un tiempo modelo a los cambios que ocurren en el [WEISS].

El agente reactivo desarrolla procesos que actúen o respondan de manera estimulo-respuesta al estado presente del ambiente en el que viven.

Es el que genera una acción de acuerdo a un cambio en el entorno, es decir, depende de que existan cambios en el entorno para realizar una acción, solo reacciona al medio y entonces efectúa tareas específicas que pueden efectuar cambios en el entorno (Ver Fig. 2.1.1).

Fig. 2.1.1, Estructura básica de un agente reactivo [WEISS].

En la Fig. 2.1.1. se puede observar que el agente reactivo tiene conocimiento del entorno y para ciertos cambios en él, tiene una colección de acciones a realizar, como un ejemplo de lo anterior se tiene:

El programa MANTA: Modeling an ANThill Activity [DAS]
Simulación de sociedades de hormigas para estudiar la emergencia del reparto de trabajo en el seno de la sociedad. Cada hormiga tiene operaciones de percepción, selección y activación que manipulan un conjunto de tareas.
Este tipo de sistemas no tiene en cuenta el pasado ni el futuro, para la toma de decisiones, solo cuenta el presente.

2.2 Métricas de Software

Es una medida que proporciona una indicación cuantitativa de la extensión, cantidad, dimensiones, capacidad o tamaño de algunos atributos de un proceso o un producto [PRESS].

La medición de un producto en cualquier empresa es parte de la especificación de dicho producto, es una forma de controlar la calidad del mismo [AIS].

Se puede medir el largo, ancho y alto de un objeto, y por comparación con un estándar la calidad sería que tanto margen de error tiene un producto, y dependiendo del uso que se le va a dar al objeto, dependerá que margen de error es permisible, también se puede medir la resistencia del producto, la durabilidad, etc. esto es fácil ya que se trata de algo tangible, un objeto que puede ser medido, pesado, etc.

En software no es tan fácil, la calidad no se puede medir contando líneas de código, complejidad, o cuantas variables se tienen, al menos no directamente, pero si se tiene un conjunto de métricas y se comparan entre ellas y con los estándares de la organización que produce el software, entonces se pueden obtener conclusiones acerca de la calidad del sistema.

Obtener una métrica de software se refiere a derivar un valor numérico para algún atributo de un producto de software (Fig. 1.2.7.3.1), un proceso de software [SOMMER].

Éstas métricas son estáticas y pueden ser de control o de predicción:

- Métricas de control: Son métricas indicadoras que se asocian con el proceso de software, se utilizan para evaluar si la eficiencia de un proceso ha mejorado, por ejemplo medir el esfuerzo y el tiempo dedicado a las pruebas de software.
- Métricas de predicción: Son métricas indicadoras que se asocian a la calidad

de producto de software, se utilizan para verificar por comparación si un producto tiene calidad de acuerdo con los estándares de la empresa.

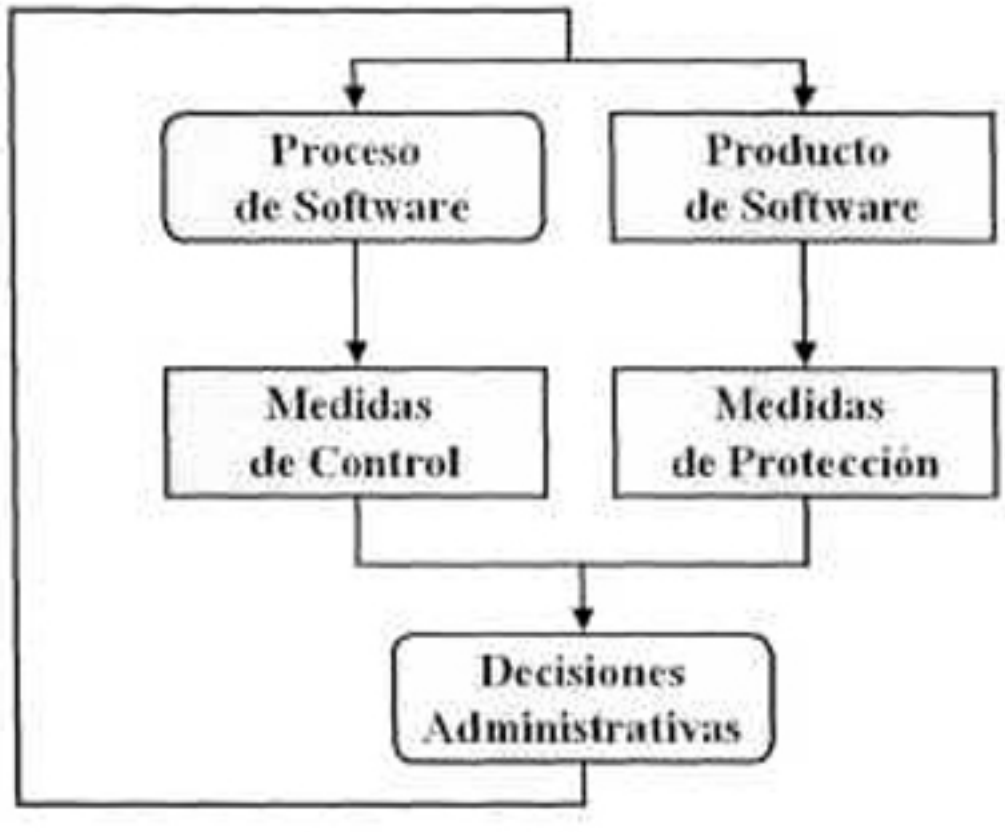

Fig. 2.2.1 Medidas de control y de predicción [SOMMER].

Ambas métricas influyen en la toma de decisiones del administrador, como se puede observar en la Fig. 2.2.1, esto representa un proceso de mejora continua, lo que quiere decir que el administrador general o de proyecto debe observar estas métricas y generar decisiones administrativas para que el mencionado proceso de mejora continua pueda ser un ciclo.

Ejemplos de decisiones administrativas son:

- Decisiones de personal.- Quienes trabajarán en que proyecto, contratación de personal, capacitación, despido, etc.
- Técnicas utilizadas en el desarrollo.- Programación por parejas, cada uno por separado, formación de equipos, programación extrema, etc.
- Costo.- Herramientas a utilizar, luz, agua, teléfono, equipo, mantenimiento, impresiones, libros, etc.
- Esfuerzo requerido.- Horas-hombre necesarias, supervisión, tiempo de entrega, etc.

En general las métricas no son utilizadas debido a que: son relativamente nuevas, no existen muchas herramientas que permitan obtener dichas métricas, la organización de procesos de software no existe o es muy pobre, o la empresa no es suficientemente madura como para implementarlos, el personal no esta capacitado para éstas o no quiere medir su proceso de desarrollo por miedo a una evaluación negativa de su desempeño [AIS].

Tener un equipo de desarrolladores es bueno, pero tener un equipo integrado y que se conoce es mejor, así pues, el obtener métricas y tener la información abierta, ayuda a conocerse y a integrar el equipo de desarrollo.

Conocer las métricas o el desempeño de una persona es de gran ayuda para conocer tiempos de entrega, es decir, se puede determinar con un mayor grado de certeza una fecha de entrega de un producto de software, cabe recalcar que es con un mayor grado de certeza, y no una certeza absoluta, por un lado trabajamos con personas, en una labor creativa y por otro lado se inicia con una estimación, de parte del personal con mayor experiencia, acerca de la complejidad y tamaño del producto a desarrollar.

Siempre que se trabaja con personal creativo se debe tomar en cuenta que son humanos y pueden cometer errores, o pueden estar cansados, estresados o con problemas, sin embargo la mayor parte del tiempo el personal se comportará de acuerdo a lo que se espera, en cuanto a rendimiento se puede esperar algo constante.

La estimación del tamaño y complejidad viene de entrevistas preliminares con el cliente, y es la única manera de conocer que desea el cliente, pero es una estimación, por más entrevistas y experiencia que se tenga, o por más definida que sea la entrevista.

Una vez hecha la estimación se propone una fecha de entrega del producto con el

sustento de conocer la cantidad de desarrolladores disponible, experiencia y grupos de trabajo, es aquí donde al conocer las capacidades de desarrollo del personal se pueden tomar decisiones con respecto de a quien delegar el desarrollo del producto, que equipos formar y en cuanto tiempo estará terminado el proyecto.

2.3 Uso de métricas

Dentro de la Ingeniería de Software las métricas son una técnica basada en medición, orientadas al proceso de desarrollo de software y sus productos para suplir oportunamente el manejo de la información y mejorar el proceso y sus productos.

Con la formación de una cultura de trabajo enfocada en la ingeniería de software, con el tiempo y la dedicación suficientes, una empresa alcanzará un nivel de maduración en el cual puede definir con más precisión su propia calidad en el desarrollo de sistemas [AIS].

Las métricas de control se utilizan para descubrir si un cambio (organizacional, de proceso, de personal, etc.) ha introducido mejoras en la eficiencia de un proceso.

Las métricas de predicción se utilizan para verificar, de acuerdo con los estándares de la empresa si un producto de software tiene calidad, en cuanto a que es fácil de mantener, de portar, si es fiable, si es usable, etc.

Cabe mencionar que una métrica NO ES UNA HERRAMIENTA DE EVALUACIÓN, aunque de cierto modo es "normal" para un administrador evaluar el desempeño del personal, ya que es parte de sus obligaciones, pero si el personal conoce que las métricas serán usadas para evaluación, el resultado de las mismas se verá afectado, ya que desearán generar código más rápido, disminuir o aumentar la

complejidad, etc. y esto contribuirá a no obtener métricas reales [AIS].

El administrador debe saber que el uso de métricas para evaluación de personal resulta contraproducente y genera ruido en las mismas [AIS].

Por ejemplo, se premiará a quien genere más líneas de código por día o por semana.

Una persona que normalmente genera líneas como:

```
Texto="\n "+LongVariables+" / "+NoVariables+" = "+(int) LongVariables/NoVariables+"\n";
```

Podrá querer generar lo siguiente:

```
Texto1="\n ";
Texto2=Integer.toString(LongVariables);
Texto3=" / ";
Texto4=Integer.toString(NoVariables);
Texto5=" = "
Texto6=Integer.toString( (int) LongVariables/NoVariables);
Texto7="\n";
Texto=Texto1+Texto2+ Texto3+ Texto4+ Texto5+ Texto6+ Texto7;
```

Ciertamente genera más líneas de código, pero no es lo que busca una empresa de desarrollo de software, de esto se infiere que dar a conocer que se evalúa al personal mediante las métricas tomadas, originará que las métricas tomadas no serán fiables.

El administrador a través de las métricas de software podrá [AIS]:

- Conocer los límites y avances "tangibles".
- Mejorar el trabajo en equipo.
- Predecir la propensión a errores del sistema desde la fase de diseño.
- Predecir el esfuerzo necesario para dar mantenimiento a un sistema.
- Obtener el grado de dificultad de prueba de un módulo.
- Predecir el esfuerzo requerido para probar las subrutinas.
- Estimar el tamaño del proyecto desde la fase de diseño.

- Proyectar el esfuerzo requerido para el desarrollo de un sistema.
- Conocer que propiedades de las métricas de software son las adecuadas para obtener calidad en un sistema.

Las métricas son una herramienta auxiliar dentro del desarrollo de software y como tal deben ser usadas, solo con la experiencia, el administrador y el equipo de desarrollo podrán conocer cuales de las métricas son las adecuadas para un sistema en particular [AIS].

2.4 Métricas estándares

En empresas de software la toma de métricas no es común, de las empresas que si toman métricas, las más utilizadas son [SMBP]:

- Métricas del esquema (76%)
- Métricas de requerimientos (67%)
- Líneas de código (62%)
- Densidad de errores (57%)
- % Riesgos del proyecto general (45%)
- Distribución de severidad de Fallar (45%)
- Productividad del programador (45%)
- % Cobertura de la prueba (44%)
- Adecuación de recursos (44%)
- Calendario, Calidad, Costo (44%)
- Puntos de función (41%)
- Movimiento Personal (36%)
- Razón Llegada/Solución de Fallas (35%)

No existe un estándar en cuanto a qué métricas son las ideales, depende con mucho de la estructura de la empresa, del personal de desarrollo, del ambiente deseado, y otros factores, si bien las métricas mencionadas previamente son las

más utilizadas, solo se recomienda el uso de unas cuantas de acuerdo a la empresa, y en éste contexto Ian Sommerville [SOMMER] señala que existen 6 métricas del producto de software:

1. *Fan In / Fan Out*: *Fan In* es una medida del número de funciones que llaman a otra función, *Fan Out* es el número de funciones que son llamadas por una función; un valor alto de *Fan In* significa que la función esta fuertemente acoplada al resto del diseño y que un cambio tendrá efectos vastos, un valor alto de *Fan Out* sugiere que la complejidad de la función podria ser alta debido a la complejidad de la lógica de control necesaria para coordinar los componentes llamados.
2. Longitud de código: es una medida del tamaño del programa, entre más grande sea el producto es más complejo y susceptible a errores será, aqui se mide el numero de caracteres en un programa, el número de lineas, el número de comentarios entre otros.
3. Complejidad ciclomática: es una medida de la complejidad de control de un programa, esta complejidad esta relacionada con la comprensión del mismo, asi como con el número de pruebas que se pueden aplicar a un producto de software.
4. Longitud de identificadores: es una medida del tamaño promedio de los diferentes identificadores en un programa, entre más grande sea el tamaño, es más probable que sea comprensible y con significado.
5. Profundidad de la condicional anidado: es una medida de la profundidad de anidamiento de las instrucciones *if* de un programa, entre más profundidad tenga más dificil de comprender será y más susceptible a errores (en potencia).
6. *Fog Index* (Índice de *Fog)*: (Niebla) es una medida de la longitud promedio de las palabras y declaraciones en los documentos. Entre más grande sea el valor del índice, el documento será más dificil de comprender.

2.5 Herramientas automatizadas

Dentro de las empresas de desarrollo de software, las herramientas más utilizadas para evaluar un producto son [SMBP]:

- (63%) *MS-Excel – Microsoft*
- (17%) *RationalSuite - Rational Software*
- (14%) *ClearQuest - Rational Software*
- (7%) QSM-SLIM – QSM
- (6%) *McCabe toolset - McCabe & Associates*
- (5%) *Function Point Workbench – Charismatek*
- (5%) Herramienta de dominio público
- (3%) *MericCenter from Distributive Software*

Es sorprendente que la herramienta más utilizada sea *MS-Excel*, de *Microsoft Corporation*, que no es una herramienta automatizada para la toma de métricas, esto implica que las métricas son realizadas manualmente, y que la información se captura en una hoja de MS-Excel y entonces se lleva un registro histórico, este sistema no es el más adecuado para evaluar un producto, ya que no está diseñado específicamente para ello.

Otras herramientas como *RationalSuite*, *ClearQuest*, etc., como ya se mencionó, son muy poco utilizadas, lo que significa que la mayoría de las empresas no tienen herramientas adecuadas para la toma de métricas.

Por otro lado, éstas métricas son obtenidas de un producto terminado, lo cual genera información adecuada para la toma de decisiones al término del desarrollo, pero las métricas durante el desarrollo generan información que se puede utilizar de manera diferente, un ejemplo simple, las líneas de código de un producto terminado pueden ser de 50,000 líneas y se tomo un tiempo de 4 meses, éstas son métricas de producto terminado en promedio se desarrollaron 411 líneas por día, pero si se toman las métricas durante el proceso de desarrollo se obtiene algo

diferente en promedio, ejemplo los lunes 387, los martes 465, los miércoles 389 líneas por día, etc. con lo que el administrador puede definir que en los días martes existe mayor productividad, al investigar el por qué, se puede obtener información que le ayude a definir estrategias para mejorar la productividad, sin llegar a sobre-explotar al personal.

3. Implementación del sistema

3.1 Sistema de métricas para el desarrollo en tiempo real de programas basado en agentes

En este documento se plantea la construcción de un sistema que continuamente tome métricas durante el desarrollo de un proyecto y almacene la información del mismo, generando a petición del usuario información histórica del proyecto, esto a través de un agente reactivo con base en la estructura de la clase Thread de Java, cabe mencionar que el agente no altera el medio en que se encuentra, solamente almacena la información del proyecto observado.

Se seleccionó el lenguaje Java JBuilder EE Ver. 3, por ser un lenguaje que permite ejecutar un programa en cualquier plataforma, es decir, cualquier persona puede utilizar el sistema sin necesidad de cambiar su plataforma de trabajo, lo que le da versatilidad y accesibilidad.

Las métricas tomadas por el sistema son:

- Longitud del código (Tamaño de la clase en bytes)
- Longitud de identificadores (Promedio)
- Líneas de comentarios (Porcentaje)
- Líneas de código
- *Fan In*
- *Fan Out*
- Registro de Archivos (clases)
- Complejidad ciclomática

Las métricas mencionadas se seleccionaron por ser las más representativas y comúnmente usadas.

Se han llevado experimentos para tratar de derivar y validar las relaciones entre

éstas métricas y, la complejidad, la comprensión y la mantenibilidad de un sistema, y de las métricas mencionadas, parecen ser los predictores más confiables la longitud del programa o del componente y la complejidad del control [SOMMER], sin embargo las organizaciones deben experimentar con varias métricas para descubrir las más apropiadas a sus necesidades.

Se menciona tiempo real ya que el sistema esta activo durante el desarrollo del proyecto en cuestión y toma métricas cuando ocurre un cambio en el mismo.

A continuación se presenta una descripción del sistema:

3.1.1 Longitud del código

Dado que las clases en lenguaje Java se guardan como archivos tipo texto, cada carácter ocupa un byte, entonces el tamaño de la clase en bytes es el número de caracteres que ocupa la mencionada clase.

3.1.2 Longitud de identificadores

Representa el número de caracteres en promedio de todos los identificadores utilizados por la clase. Si bien siempre existen variables en un programa, estas no siempre representan por el nombre, el tipo de valor que conllevan, es decir, una variable entera con el nombre "dato", no significa mucho para el programador que no conozca el sistema, pero en una empresa de software, donde no siempre el mismo programador estará a cargo del sistema la variable debe tener un nombre significativo, por ejemplo, "Contador", esto es igual para clases, métodos y variables, que tan larga será la longitud del identificador depende del administrador y de los desarrolladores.

3.1.3 Líneas de comentarios

Representa el porcentaje de líneas de comentarios, entre líneas de código. Si bien un porcentaje del 10% de líneas de comentarios es el adecuado para un sistema

cualquiera [AIS], esto no siempre sucede, pero al llevar a cabo métricas con el sistema en proceso, el desarrollador puede ayudarse a obtener un 10% ideal.

3.1.4 Líneas de código

Representa el número de líneas de código por clase, donde cada linea contiene una instrucción simple. Entre más lineas tenga una clase, más compleja y difícil de mantener será, además de que existirán más errores en el mismo y representa más tiempo de corrección de esos mismos errores, pero éste número depende de la capacidad y experiencia de los desarrolladores.

3.1.5 *Fan In*

Representa la cantidad de veces que un método es llamado por otro método de la clase o de otras clases. También representa el grado de ensamble que tiene un sistema, es decir, si éste número es grande, un cambio en ésta clase significa una alteración en el sistema en total.

3.1.6 *Fan Out*

Representa la cantidad de veces que un método llama a otros métodos, de la misma clase y de otras clases. Igual que la anterior, representa al grado de ensamble que tiene el sistema, y cualquier alteración significa que el sistema completo puede cambiar.

3.1.6 Registro de archivos (clases)

representa la presencia o ausencia de clases en el sistema, es decir, si una clase es creada en el dia 15 del proyecto se puede observar este cambio, de igual manera si una clase es eliminada del sistema, queda el registro histórico de la misma.

3.1.7 Complejidad Ciclomática

Para la complejidad ciclomática se utilizó el siguiente método, que si bien no está demostrado plenamente, se han comparado resultados manuales y automáticos, y

a la fecha no se encontrado discrepancia alguna, cabe mencionar que la sencillez del método hace posible el calculo automático de cualquier algoritmo, no importando las demás instrucciones o el anidamiento.

La complejidad ciclomática se calcula de la siguiente manera (Fig. 3.2.1):

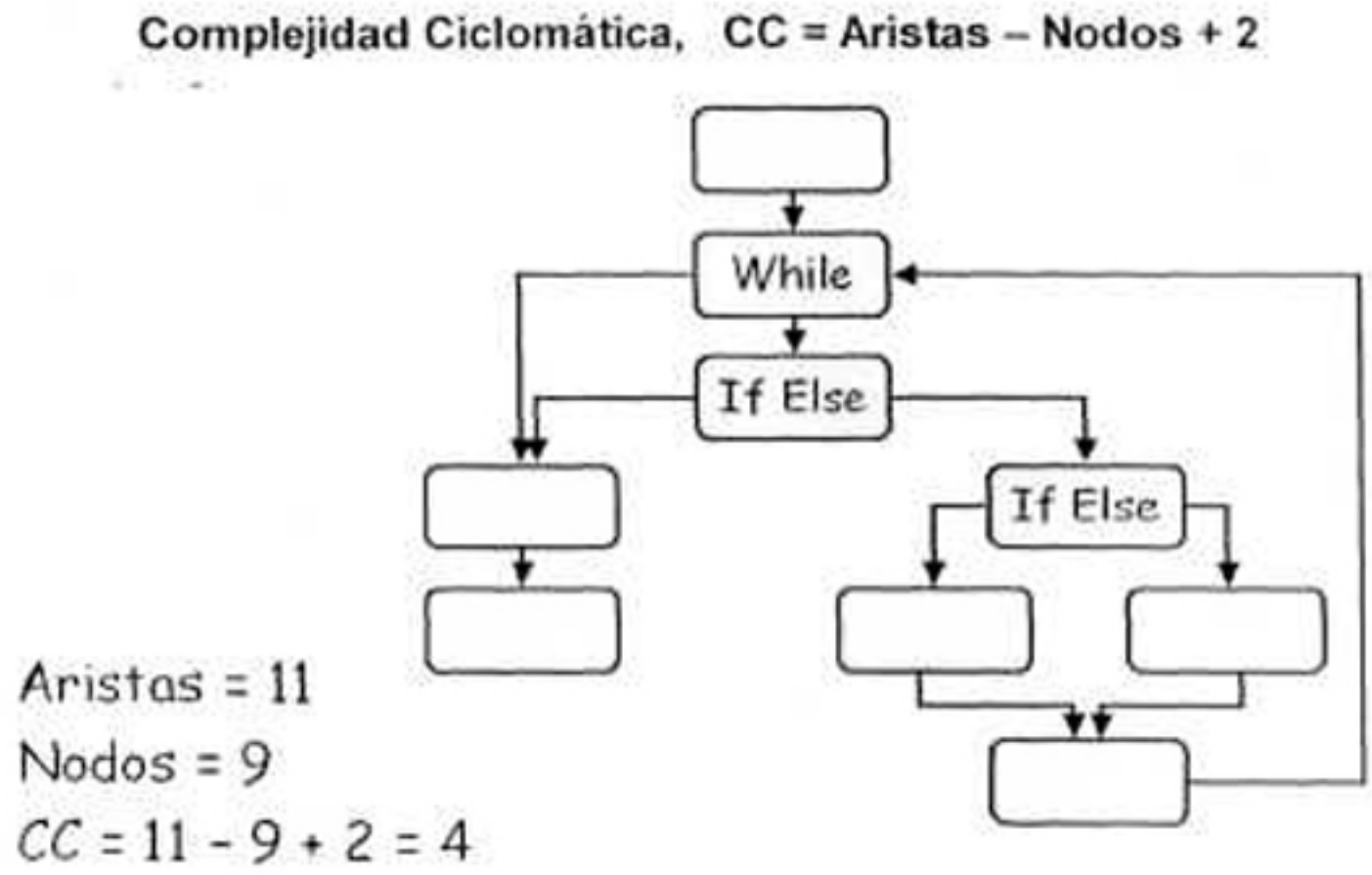

Fig. 3.2.1, Cálculo de complejidad ciclomática, CC

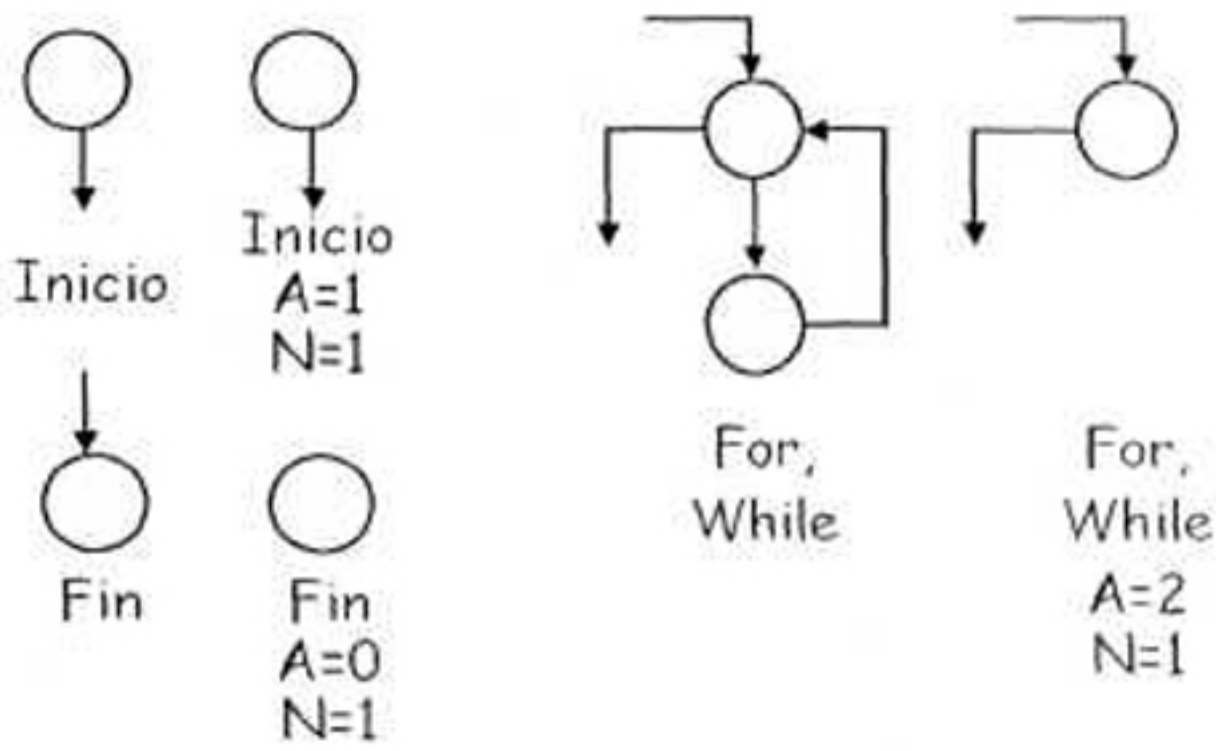

Fig. 3.2.2, INICIO, *FIN*, *WHILE*, con su valor para el método de CC

Se propone entonces, a través de la observación y el análisis, que los métodos en forma individual sean modificados durante el análisis y cada nodo no tenga la arista de llegada, la cual será tomada en consideración como la arista de salida del nodo anterior, quedando entonces la instrucción Fin sin aristas de llegada o salida (ver Fig. 3.2.2.), también que cualquier otra instrucción que no sea de control (ver Fig 3.2.4), no sea tomada en cuenta ya que será anulada por la forma de obtener la complejidad ciclomática, al tener un Arista y un Nodo, lo mismo que el interior de las instrucciones *For*, *While* y *Do*, como se puede observar en las figuras 3.2.2, 3.2.3, 3.2.4, y 3.2.5.

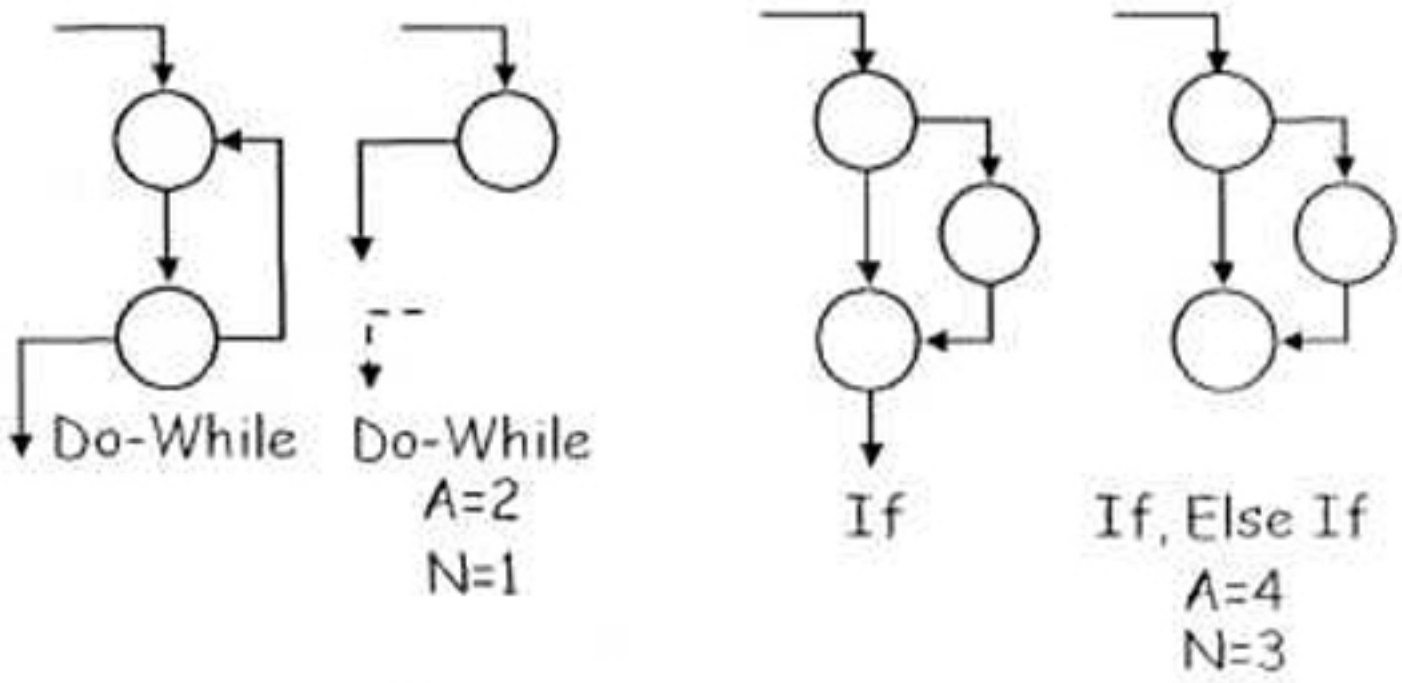

Fig. 3.2.3, *DO–WHILE*, *IF*, *ELSE–IF*, con su valor para el método de CC

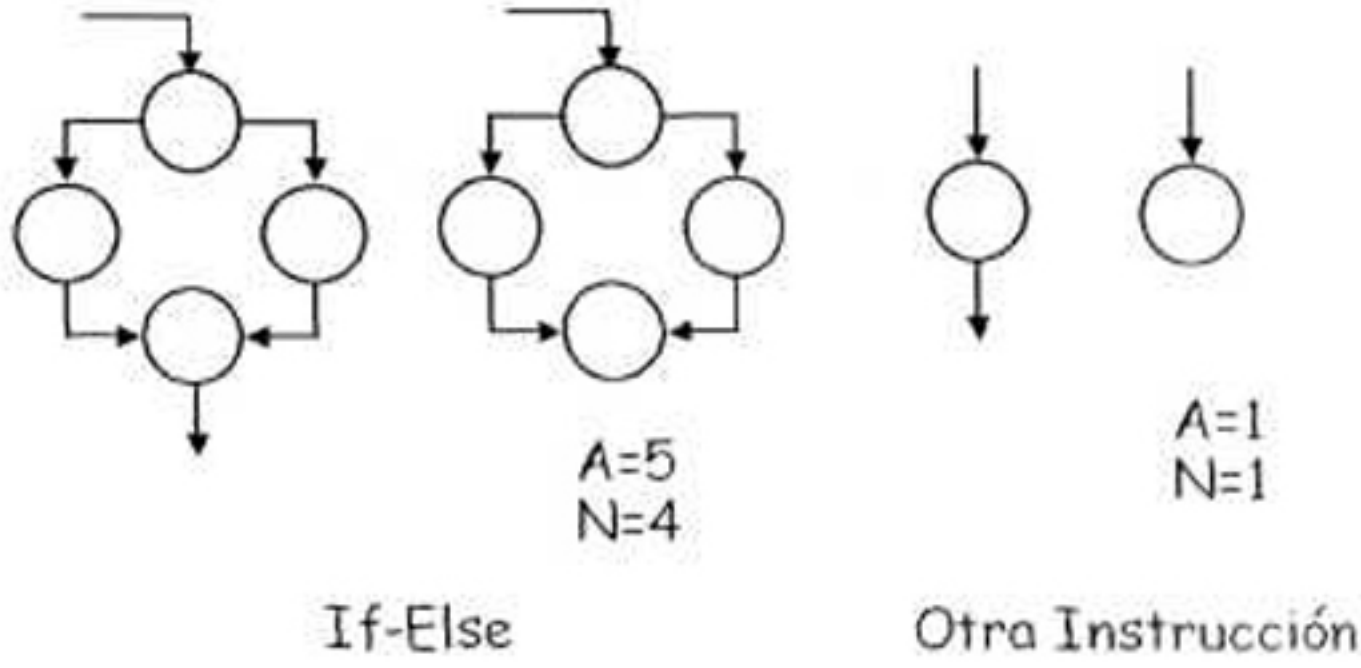

Fig. 3.2.4, *IF-ELSE*, Cualquier otra instrucción, con su valor para el método de CC

Cabe observar que la instrucción *IF* debe ser separada de la instrucción *IF-ELSE*, ya que los valores son diferentes, Fig. 3.2.3 y Fig. 3.2.4.

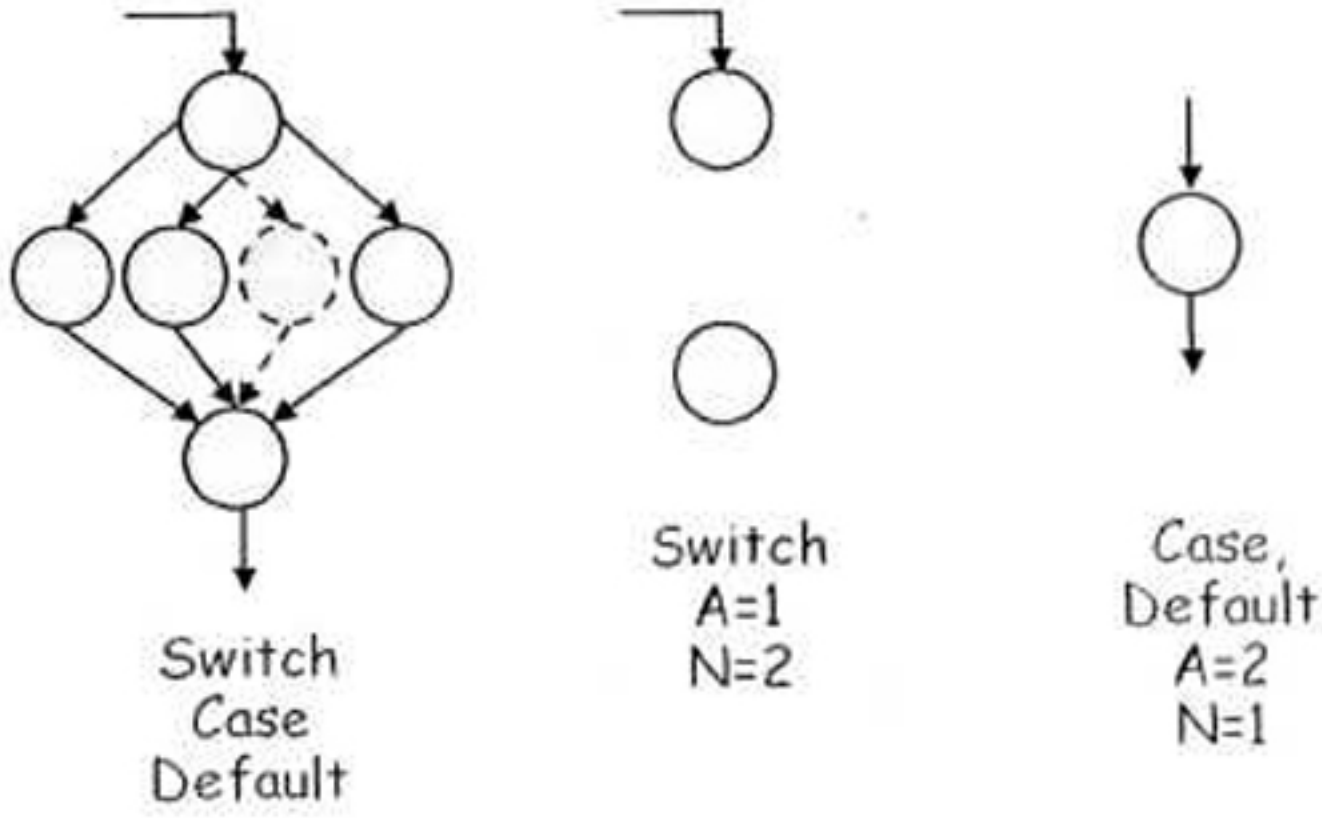

Fig. 3.2.5, *SWITCH–CASE–DEFAULT* con su valor para el método de CC

Aristas	Nodos	Descripción
1	1	Inicio
0	1	Fin
2	1	For, While, Do-While
1	2	Switch
2	1	Case, Default (Switch)
4	3	If
5	4	If-Else

Fig. 3.2.6, Tabla de valores para el método de CC

En la tabla de la figura 3.2.6, muestra de que manera serán tomadas en cuenta

cada instrucción de control para obtener la complejidad ciclomática, simplemente se cuentan cuantos *IF*, cuantos *FOR*, cuantos *IF-ELSE*, etc., y se suman para cada uno, los valores de Aristas y Nodos, deben de sumarse además los valores de INICIO y de FIN para cada proceso y se efectúa la fórmula ya mencionada:

Complejidad Ciclomática, CC = Aristas – Nodos + 2

4.- Resultados

Cualquier proyecto al desarrollarlo cambia, se hace más grande, complejo etc., con la toma de métricas se puede observar ese cambio (por ejemplo, observando como crece el número de líneas por día, semana, proyecto, etc.), y a través de la experiencia un administrador puede determinar las métricas que deben ser tomadas en cuenta para sus necesidades particulares o de la empresa.

Las métricas tienen significado solo si han sido examinadas para una validez estadística. El gráfico de control es un método sencillo para realizar esto y al mismo tiempo examinar la variación y la localización de los resultados de las métricas [PRESS].

4.1 Sistema medido

Los ingenieros de software y sus gestores pueden obtener una visión más profunda del trabajo que realizan y del producto que elaboran creando una base de datos que contenga mediciones del proceso y del producto [PRESS].

Para tomar métricas y llevar a cabo un registro histórico se debe tener el sistema de toma de métricas (Centinela), pero debido a que no se contaba con el sistema terminado, no se tomaron métricas durante el desarrollo del mismo.

Debido a esta situación sólo el presente proyecto fue medido gracias a que se guardó un respaldo cada cierto tiempo (ver Fig. 4.1.1), aunque se trató con constancia, las versiones en un proyecto de ésta magnitud son tremendas, por cada hora trabajada en el desarrollo de éste sistema se guardaban los archivos de 4 a 6 veces, si se trabajó un promedio de 4 horas diarias durante 190 dias se deberian tener 3800 métricas tomadas, sin embargo las métricas almacenadas solo son 56, lo que otorga un respaldo del desarrollo cada 3.4 dias aproximadamente.

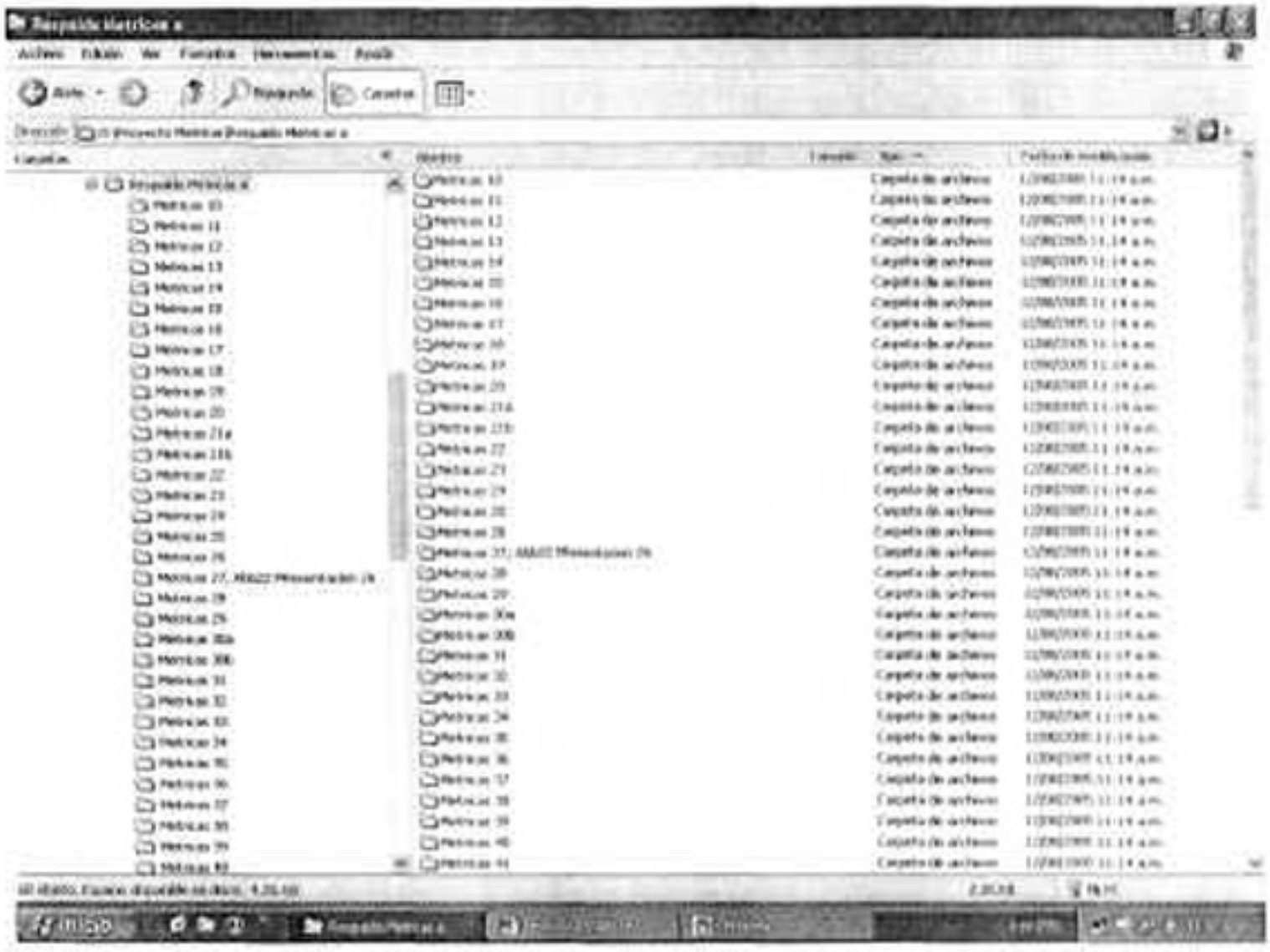

Fig. 4.1.1 Respaldo histórico de las clases

4.1.1 Lineas de Código

Aún con éstas restricciones, se observan cambios en el desarrollo del sistema, un ejemplo sencillo es el número de lineas por clase, se nota en la Figura 4.1.2, que existen incrementos y decrementos en la cantidad de lineas por clase.

La interpretación de la Figura 4.1.2, (Ver Fig. 6.3.19) es que, durante el desarrollo del sistema se han aumentado métodos por cada clase y se han quitado otros métodos, debido al desarrollo mismo del sistema, además se observa que algunas clases, inician su existencia un poco antes que de la mitad del proyecto, esto debido a que en el proyecto se tuvo un cambio drástico, se eliminaron clases (Reg_Directorio, Ventana, CArchivo y CDemon) y se crearon otras (Directorio, cArchivo y cDaemon), la clase Ventana se elimina definitivamente del proyecto.

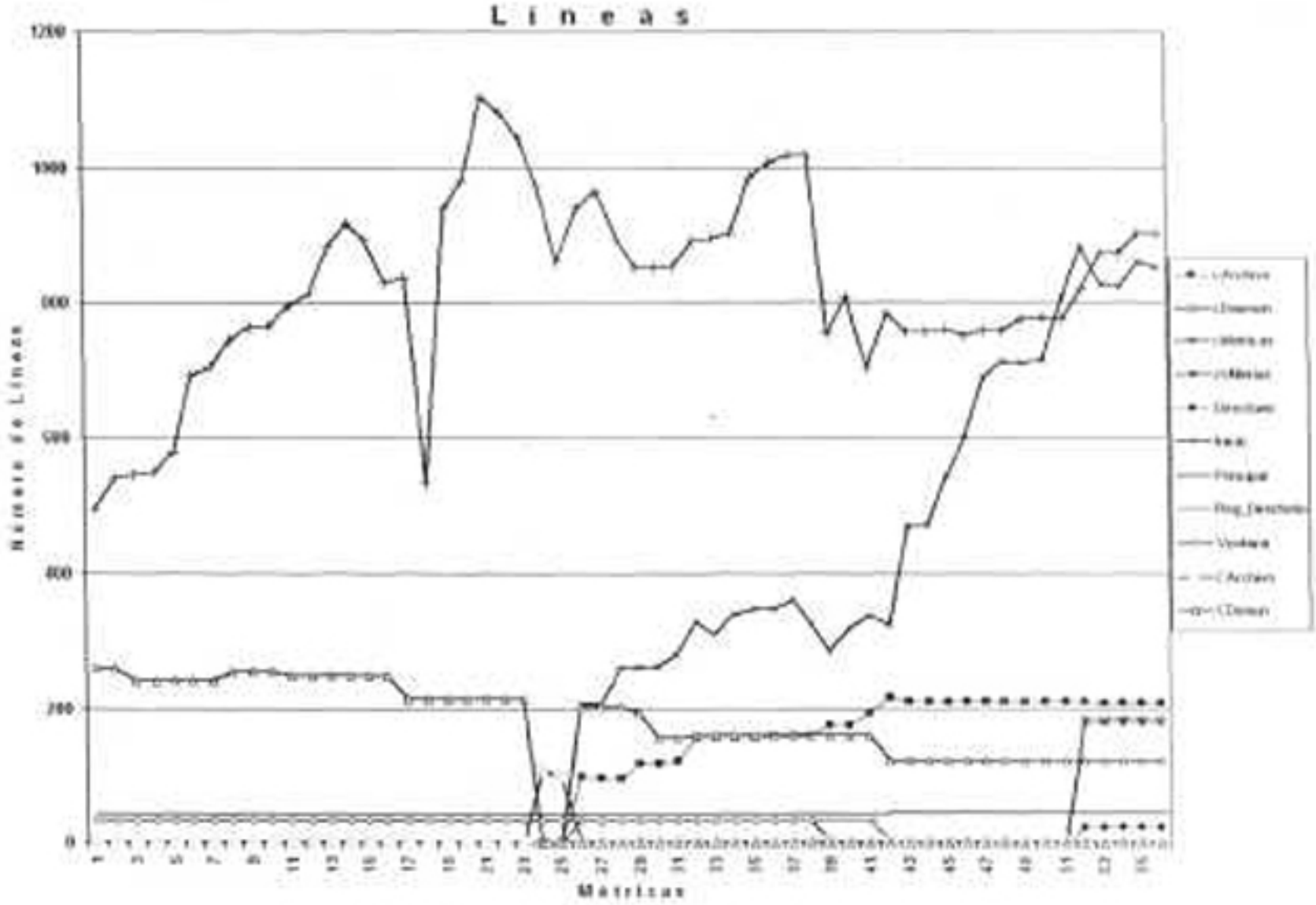

Fig. 4.1.2 Registro histórico de lineas por clase.

Se observa que la clase cDaemon tiene un incremento constante, lo que es un desarrollo normal, aún con los altibajos observados, pero la clase Inicio (parte superior de la Fig. 4.1.2) si bien tiene un incremento desde el inicio hasta el final, los cambios son importantes y muchos (11 cambios), lo que implica que la clase Inicio ha sido reconstruida o al menos alterada drásticamente varias veces.

4.1.2 Registro de archivos

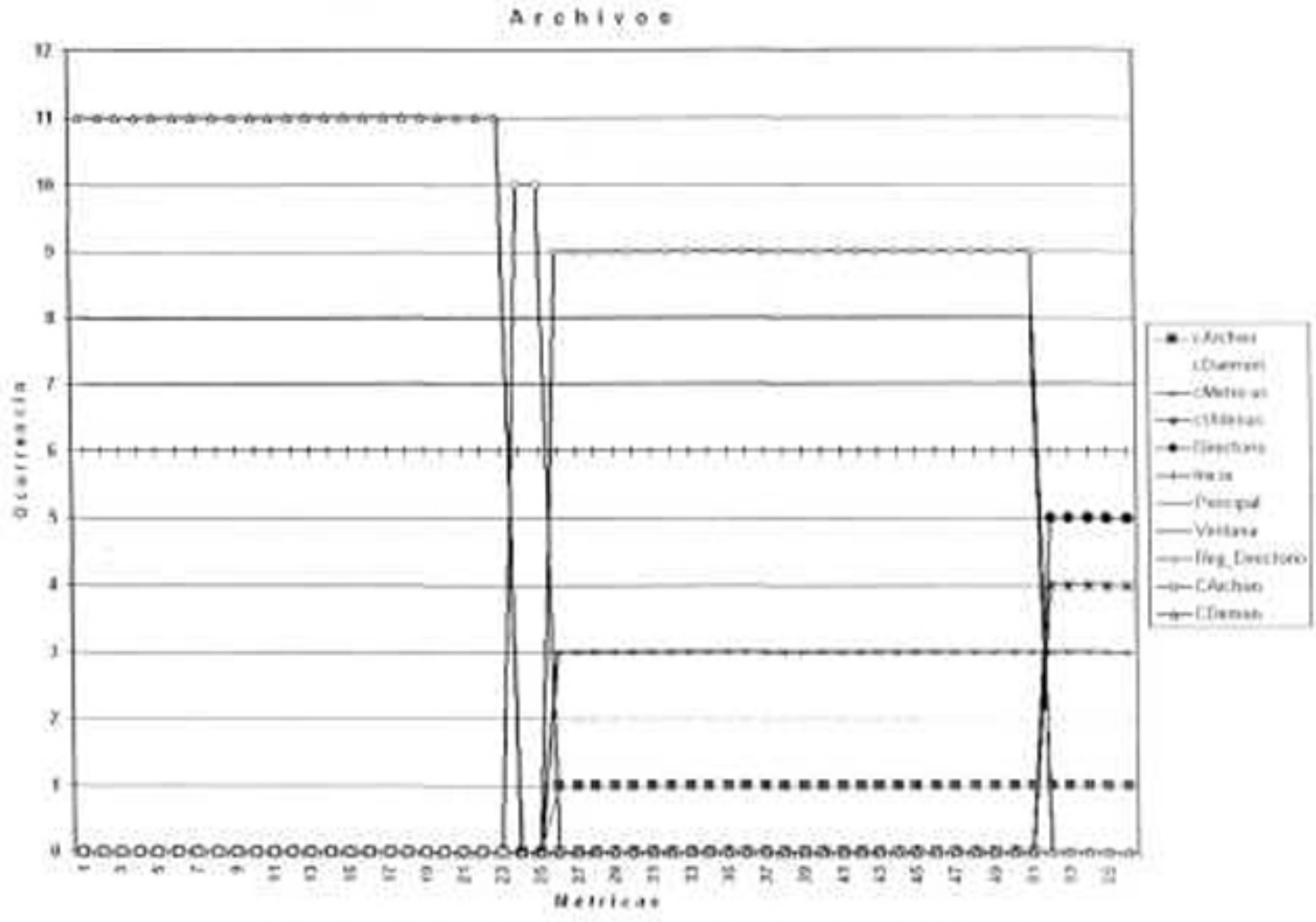

Fig. 4.1.3 Registro histórico de Archivos (Clases).

En la figura 4.1.3, se observa la existencia o no (1 o 0), de una clase en el proyecto (Ver Fig. 6.3.7), durante el desarrollo del mismo, a simple vista se observa en que momento una clase empieza a existir y en que momento se retira del proyecto.

Se observan qué clases se originan al comienzo del proyecto, cuáles se eliminan durante el proyecto y qué clases son creadas en el medio.

Debido a que se empalman los colores, se decidió a cambiar los valores de cada clase para fines de observación, es decir, los valores del sistema son 1 y 0, para poder gráficar se cambiaron los valores 1 por valores entre 1 y 11, ya que son 11 clases.

4.1.3 Líneas de Comentarios

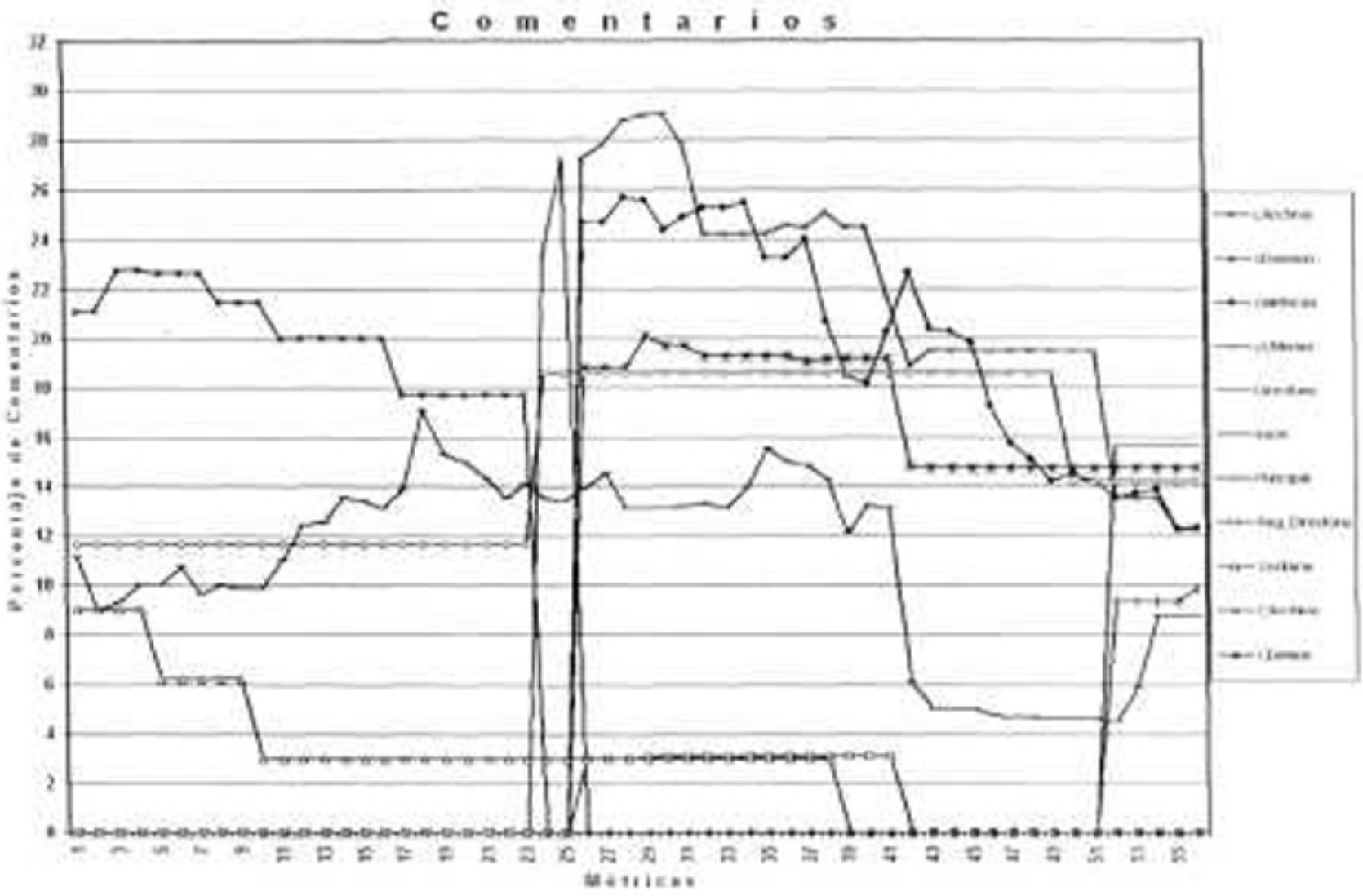

Fig. 4.1.4, Registro histórico de porcentaje de comentarios por clase

En la Fig. 4.1.4, se tiene el registro histórico de porcentaje de comentarios por clase (Ver Fig. 6.3.9). Si bien el porcentaje de comentarios recomendable es el 10%, [AIS], se observa que durante el desarrollo del proyecto se tiene hasta un 28%, pero hacia el final del proyecto se ve que la tendencia clara hacia el 10%, (entre 8% y 16%), esto debido a un esfuerzo del desarrollador, al conocer sus resultados, se deduce entonces que es necesaria una mayor conciencia al generar un comentario, de manera tal que se acerque al 10%, sin embargo es necesario mencionar que éste 10% es una recomendación, y el administrador del proyecto debe otorgar cierta flexibilidad a los desarrolladores acerca de éste punto.

La tendencia clara hacia el 10% al final del proyecto, menciona un esfuerzo conciente por lograr el 10%, pero también puede indicar que el nivel de complejidad de las clases es alto, por lo que un mayor porcentaje de comentarios por clase, puede ser recomendable, esto a discreción del desarrollador.

4.1.4 Longitud de identificadores (Promedio)

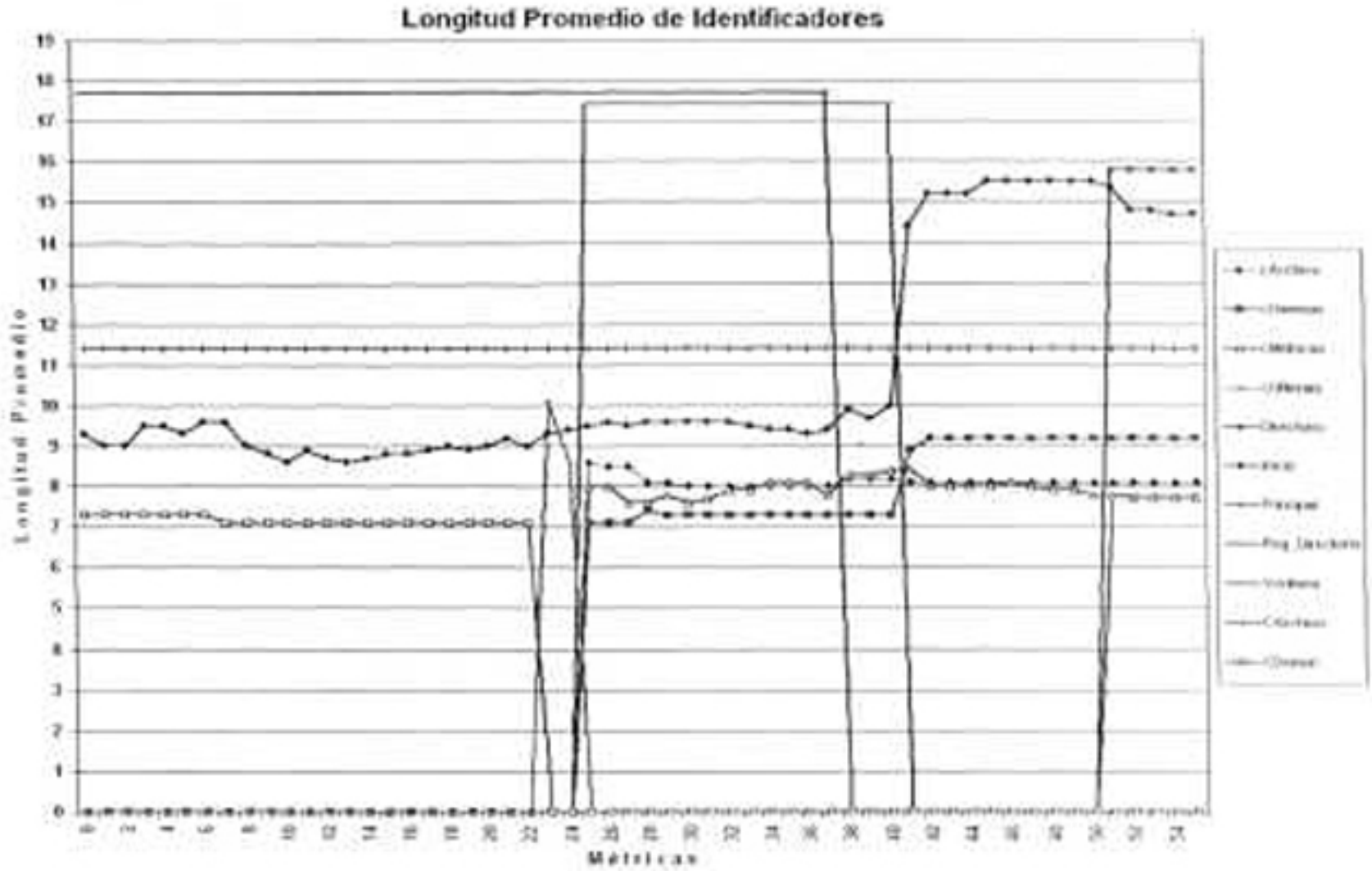

Fig. 4.1.5, Longitud promedio de identificadores por clase.

El la Fig. 4.1.5, se tiene la Longitud promedio de identificadores por clase, obtenida del registro histórico (Ver Fig. 6.3.13). Si la longitud promedio de los identificadores es alta, se indica que el nivel de comprensión por parte del desarrollador y algún otro participante del equipo de trabajo, es alto, sin embargo, nuevamente se recomienda al administrador o al supervisor que a través de la experiencia se establezca una longitud aceptable para la empresa o el equipo.

Se observa que la clase Principal es constante en todo el proyecto, esto debido a que ésta clase es generada automáticamente por la plataforma JBuilder EE Ver. 3, desde el inicio del mismo y no es alterada por el desarrollador.

Encontrar un equilibrio entre la longitud de identificadores y la facilidad de capturar código, depende en mucho del desarrollador, pero es el administrador el que establece los parámetros, sin embargo la complejidad del programa afecta en mucho la longitud promedio de identificadores.

4.1.5 Longitud del código

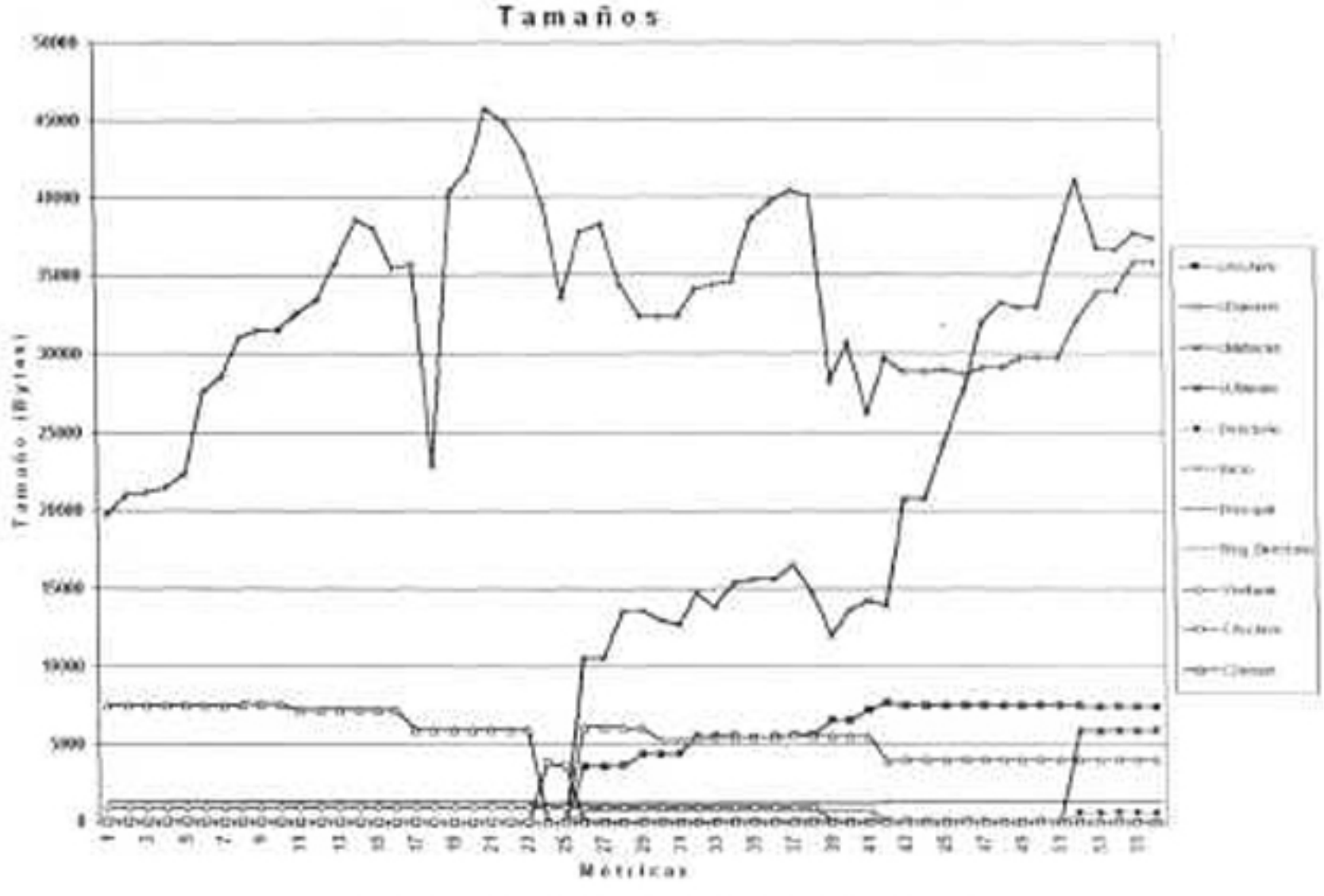

Fig. 4.1.6, Tamaños de clases (Bytes)

En la Fig. 4.1.16. se tienen los tamaños de clases, obtenidos del registro histrórico de tamaños de clases (Ver Fig. 6.3.11). Debido a la existencia de cambios drásticos durante el desarrollo del proyecto, se observan incrementos y decrementos en el desarrollo, estos cambios drásticos pueden significar que existieron problemas de comprensión, o de estructura, e incluso retrocesos, en este caso fue un cambio en la estructura del sistema, ya se ha mencionado que algunas clases dejaron de existir y otras fueron creadas durante el transcurso del desarrollo, se implica entonces una falta de previsión para éste sistema.

Cualquier proyecto tiene una tendencia a crecer, poco o mucho, pero cualquier reducción del tamaño implica la eliminación de instrucciones, ya sea por ser obsoletas o por encontrar una forma mejor de realizar las tareas.

4.1.6 Complejidad ciclomática

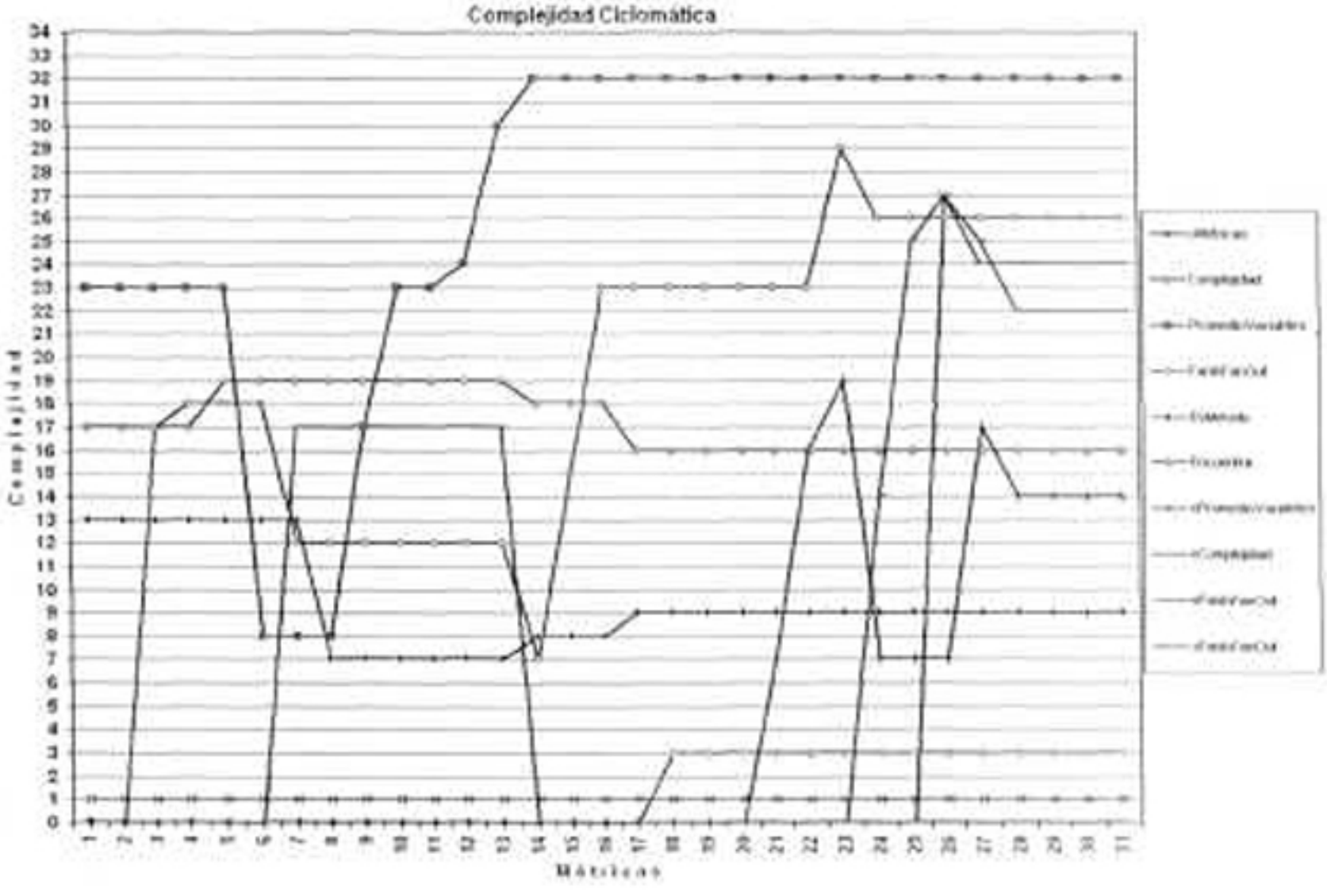

Fig. 4.1.7, Complejidad Ciclomática, Clase cMetricas.

En la Fig. 4.1.7, se tiene la complejidad ciclomática obtenida del registro histórico de la complejidad ciclomática de la clase cMetricas (Ver Fig. 6.3.15). No todos los métodos de la clase cMetricas, están representados en la gráfica, debido a que eran demasiados para una sola gráfica, los métodos son representativos de la clase, además se observa que existen solamente 31 tomas de métricas, debido a que, en las primeras 24 tomas, esta clase no existe, por lo que los datos de cada método están en ceros.

En general se observa una propensión a aumentar la complejidad de cada método de la clase cMetricas, esto debido a que los métodos más complejos se dejaron para desarrollar al final del proyecto, sin embargo también se observan cambios en la complejidad, (Fig. 4.1.7., Método PromedioVariables) un vez más la razón es que hubo un cambio en la estructura del proyecto, pero también se debe a que algunos métodos fueron simplificados.

4.1.7 *Fan In*

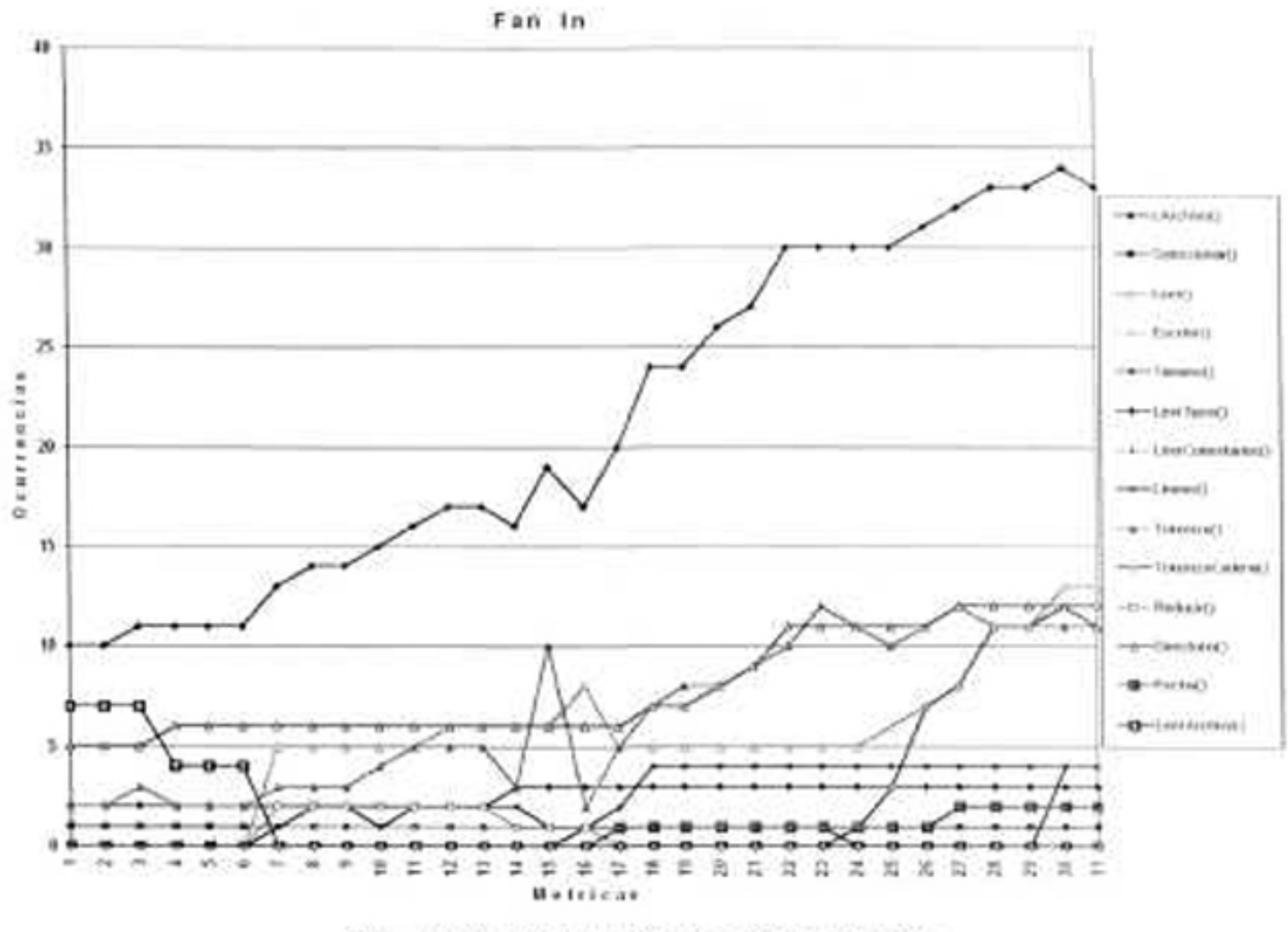

Fig. 4.1.8, *Fan In* de la clase cArchivo.

En la Fig. 4.1.8. se observa el *Fan In* de la clase cArchivo, obtenida del registro histórico de *Fan In*, (Ver Fig. 6.3.17). *Fan In* es la cantidad de llamadas a los métodos desde algún otro método y/o clase.

Se observa entonces que existen mas llamadas hacia el final del proyecto, lo que significa que el proyecto se integra más conforme avanza, también que nuevos métodos están siendo implementados y son llamados por otros, además se observa que existen solamente 31 tomas de métricas, debido a que, en las primeras 24 tomas, esta clase no existe, por lo que los datos de cada método están en ceros.

Un método altamente integrado es deseable pero significa también que un cambio en un método repercute en gran manera en la clase y/o el proyecto.

En la gráfica, la clase cArchivo se integra más conforme avanza el proyecto lo cual es normal, pero también se observa que los cambios en la estructura no afectaron mucho a esta clase, lo que significa que ésta clase esta bien desarrollada pero no muy integrada con las demás clases, ya que esta es una clase auxiliar.

4.1.8 *Fan Out*

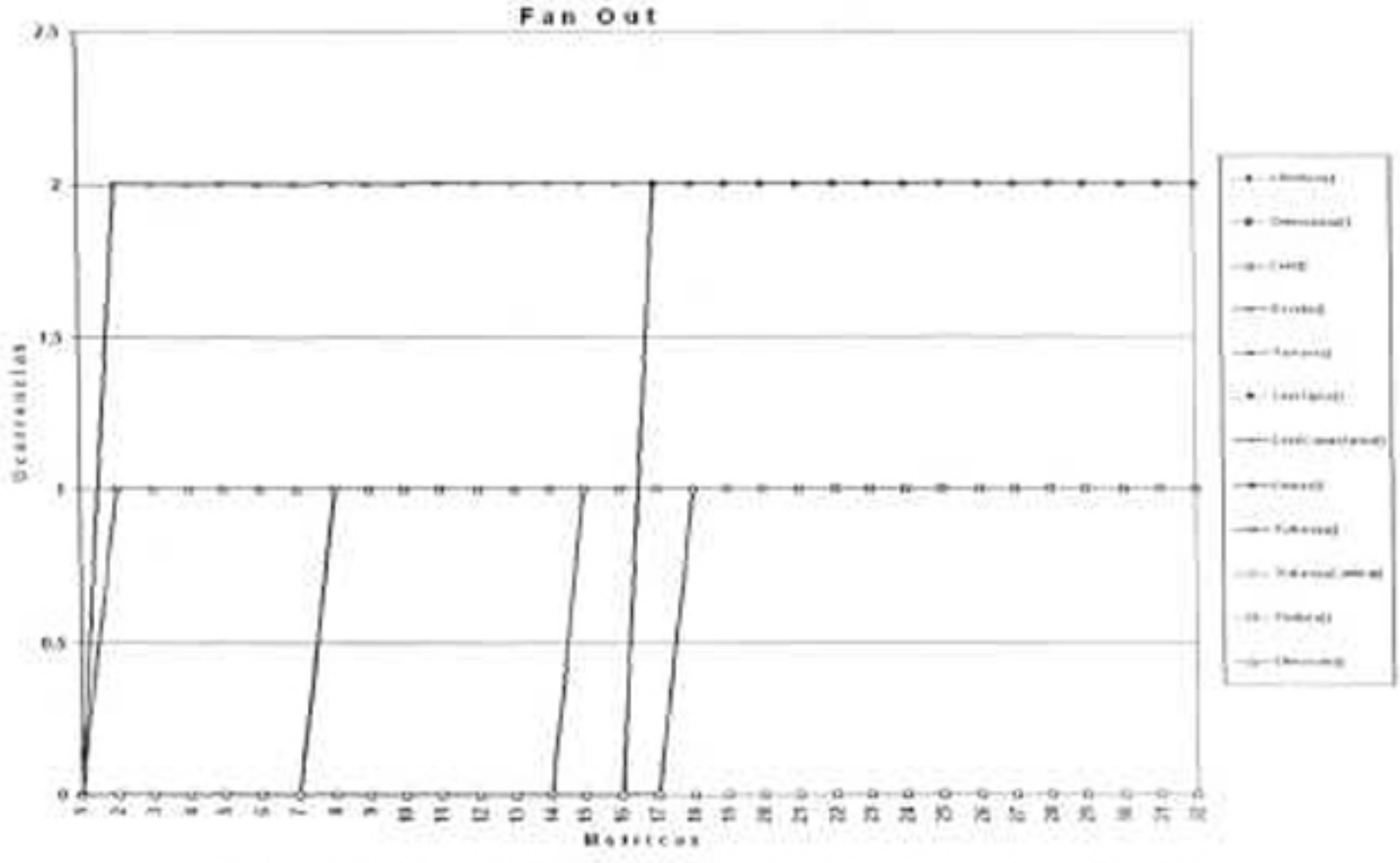

Fig. 4.1.9, *Fan Out* de la clase cMetricas, Ver Fig. 3.2.23

En la Fig. 4.1.9. se observa el *Fan Out* de la clase cArchivo, obtenida del registro histórico de *Fan Out*, (Ver Fig. 6.3.17). *Fan Out* se refiere a cuantas veces un método llama a otros, en su clase o en otras clases, nuevamente un número alto, significa que el proyecto esta altamente integrado.

Se observa entonces nuevamente que existen mas llamadas hacia el final del proyecto, lo que significa que el proyecto se integra más conforme avanza, además se observa que existen solamente 31 tomas de métricas, debido a que, en las primeras 24 tomas, esta clase no existe, por lo que los datos de cada método están en ceros.

Los cambios en el índice de *Fan Out*, significan que los cambios en la estructura del proyecto si afectaron a ésta clase, pero no a todos los métodos, lo que sugiere que esta clase no esta altamente integrada al resto del proyecto, principalmente debido a que esta es una clase auxiliar.

5.- Conclusiones

- Las métricas son una excelente técnica para el desarrollo de software.
- Fomentar una cultura de trabajo sobre la base de la toma de métricas es una forma excelente de lograr calidad y mantenerla.
- Las métricas a tomar en cuenta por parte de los administradores, supervisores y desarrolladores, deben elegirse basándose en la estructura propia de la empresa, tomando en cuenta el tipo de organización de personal, y sus capacidades.
- No todas las métricas mencionadas deben tomarse en cuenta para el desarrollo de software, lo recomendable es tres a cinco métricas que sean relevantes a la empresa o para la evaluación de un proceso de desarrollo.
- El centinela otorga al supervisor tiempo para dedicarse a otras actividades, ya que es el agente el que verifica continuamente si existen cambios en el sistema observado.
- La interpretación de resultados debe basarse en la observación de situaciones durante el proceso y en la experiencia del supervisor.

5.1.- Trabajo futuro

- Aumentar el número de métricas para otorgar más herramientas al usuario.
- Generar interpretaciones para las métricas, aunque sea de manera general.
- Agregar una base de datos para diferentes lenguajes, para que el sistema sea más versátil.
- Agregar inteligencia al agente para que tome decisiones, ejemplo: si una clase es modificada en su fecha y hora pero no en su contenido, (Salvar Proyecto), observar si existe cambio en las métricas, si no existe no agregar las métricas tomadas.

- Crear un sistema multiagente, que controle a través de un solo sistema varios proyectos, o áreas de un mismo proyecto grande pero separado, ejemplo: trabajo separado por equipos en una empresa de software.
- Almacenamiento de información estadisticamente hablando, si bien la información se guarda, la creación de información estadistica puede ayudar a comprender la orientación del proyecto y del personal.
- Generar una base de datos, donde se pueda almacenar la información de una empresa, organizada por proyectos, de aqui obtener información estadistica ya procesada.
- Implementar rutinas para observar la información en gráficas.
- Orientar el sistema a Aspectos, que le daría mayor amplitud y calidad al sistema en si mismo.

6. Anexo A. Código del Centinela de Métricas para el desarrollo de software

6.1 Código de clases relevantes

Ver Disco Anexo

6.2 Descripción del sistema

En la clase **cArchivos.java** se localizan los métodos que permiten el acceso a los archivos tipo texto observados, los nombres de los métodos principales permiten conocer que efectúa cada método: Seleccionar(), LeerArchivo(), TamanoArchivo(), Directorio(), Toqueniza().

La clase **cDaemon.java**, es el agente del sistema y se realiza al extender la clase Thread de Java, que tiene el método *run*, que es la base de la clase, gracias a éste método, se puede verificar el entorno cada cierto tiempo, establecido en el método *run* por la instrucción *sleep()*, la reactividad del agente consiste en actuar solamente cuando ocurre un cambio en la fecha y hora de alguno de los archivos observados, los métodos principales de esta clase son: *main()*, *run()*, Tiempo(), Verifica().

La clase **cMetricas.java**, es donde se encuentran los métodos que efectivamente toman las métricas del proyecto, para esto se envia el nombre del archivo a medir con la ruta incluida, los métodos principales de esta clase son: Complejidad(), PromedioVariables(), *FanInFanOut*(), Longitud(), Comentarios()

La clase **cUtilerias.java**, contiene rutinas auxiliares generales para el sistema, éstas son utilizadas por las otras clases para tareas rutinarias o repetitivas.

La clase **Directorio.java**, es una clase auxiliar del método Directorio(), en la clase cArchivo, permite a Directorio() seleccionar únicamente los archivos con extensión específica, si no existiera esta clase en el proyecto, se seleccionarían todos los archivos del directorio.

La clase **Inicio.java**, es la interfaz con el usuario y permite a través de botones que el usuario trabaje con el sistema, seleccionando, imprimiendo, u observando los resultados que se muestran continuamente en una ventana *JtextArea*, entre otras acciones.

En la clase **Principal.java**, básicamente genera la ventana de interfaz centrándola en la pantalla.

6.3 Manual de Usuario del Centinela

Dentro del proyecto de métricas para el desarrollo en tiempo real basado en agente, se desarrolló un sistema de nombre Centinela, que cumple con éstas expectativas.

El sistema principia con una ventana de nombre Centinela como se observa en la Fig. 6.3.1, donde se puede iniciar el sistema, ver tipos de variables, maximizar la ventana o salir del sistema.

Fig. 6.3.1 Ventana principal

Al iniciar el Centinela (ver Fig. 6.3.2) se selecciona cualquier archivo y el sistema selecciona todos los archivos con extensión java que existen en el directorio seleccionado, verifica su fecha de ultima modificación y almacena esta información para verificación futura, ésta se realiza cada 5 segundos, pero puede cambiar el intervalo aumentando o disminuyendo por segundos o minutos.

Fig. 6.3.2 Seleccionar archivo

Posteriormente en la ventana se muestran los archivos seleccionados y el Centinela comienza la verificación - si algún archivo cambia su fecha de modificación entonces se toman las métricas del proyecto y se almacenan (ver Fig. 6.3.3).

Fig. 6.3.3 Archivos observados

El botón Tipos permite observar los tipos de variables definidas para el sistema y cambia por el botón Añadir, éste permite aumentar nuevos tipos de variables, escribiendo en la ventana de edición los nuevos tipos y oprimiendo el botón Añadir (ver Fig. 6.3.4).

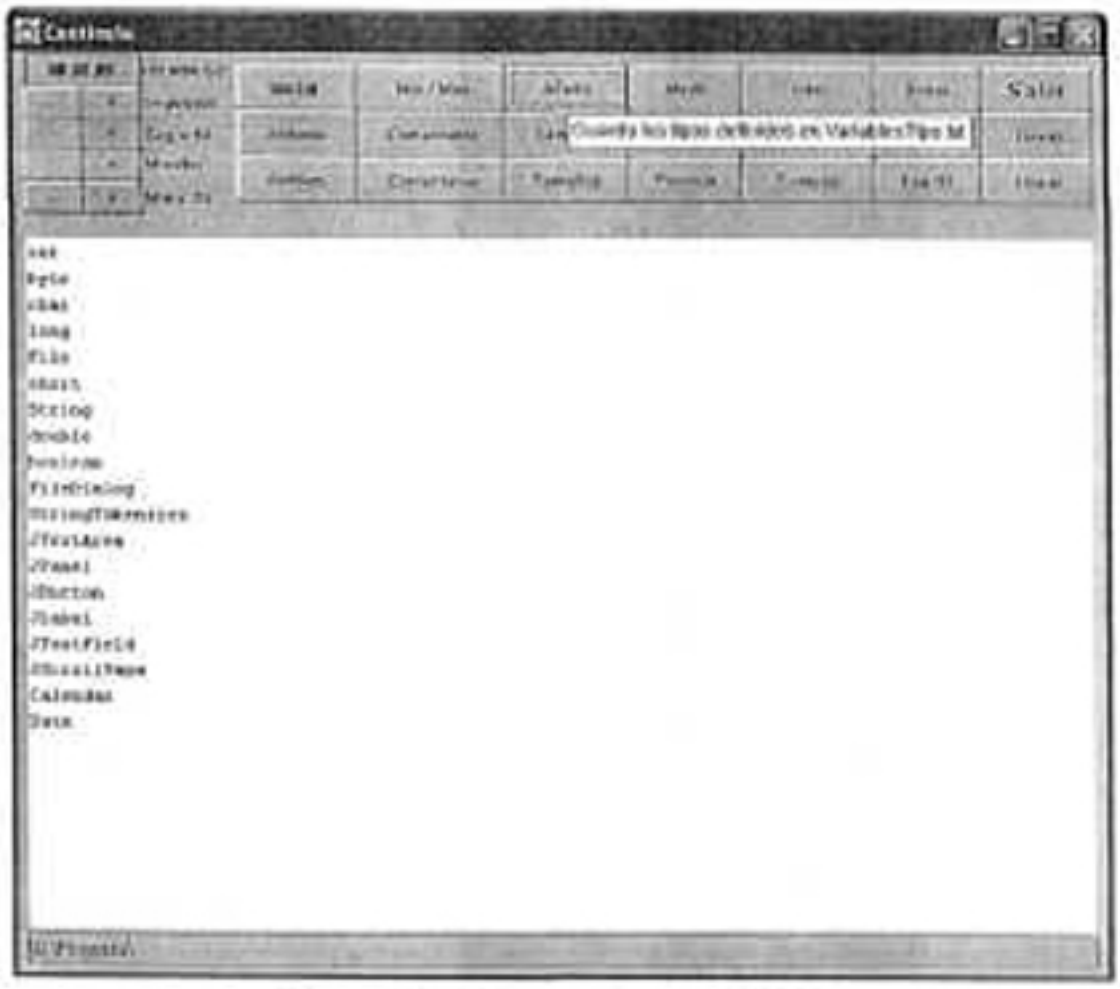

Fig. 6.3.4 Tipos de variables

En la Fig. 6.3.5 se observa la toma de métricas directa mediante el botón Medir, esta información es almacenada en un archivo con extensión mtr, el nombre se toma del archivo más actual, y representa la información de la fecha de modificación en formato long (fecha en milisegundos), en el ejemplo el archivo se llama "00000112406962794l.mtr".

Fig. 6.3.5 Métricas tomadas por el usuario

Una vez hecho lo anterior se pueden tomar métricas directamente con los botones:

Archivos	Permite conocer los nombres de los archivos del proyecto.
Comentarios	Muestra el porcentaje de comentarios por clase.
Tamaño	Muestra la longitud de código (bytes de cada clase).
PromVar	Muestra la longitud promedio de identificadores por clase.
Complej	Muestra la complejidad ciclomática por método por clase.
Fan IO	Muestra el Fan In y el Fan Out por método por clase.
Lineas	Muestra el numero de líneas de cada clase.

Los botones con letra azul permiten conocer las métricas del proyecto (los archivos con extensión java), que son las clases que están siendo modificadas por el usuario durante el desarrollo del proyecto, y los botones con letra roja permiten conocer el historial de las métricas (los archivos en el mismo directorio del proyecto con extensión mtr) que son los archivos tipo texto que almacenan la información de cada métrica tomada durante el proyecto.

La diferencia es que los botones azules presentan información de las clases existentes en el directorio seleccionado y los botones rojos presentan información acumulada durante el proyecto, cabe recordar que, cada vez que una clase es modificada y salvada, la fecha cambia y son tomadas las diferentes métricas del proyecto, se presenta a continuación la información de métricas y del historial de métricas de un proyecto (el mismo sistema fue almacenado por etapas y evaluado posteriormente), los archivos actuales del proyecto y el historial de archivos (clases) usados del proyecto (ver Fig. 6.3.6 y Fig.6.3.7).

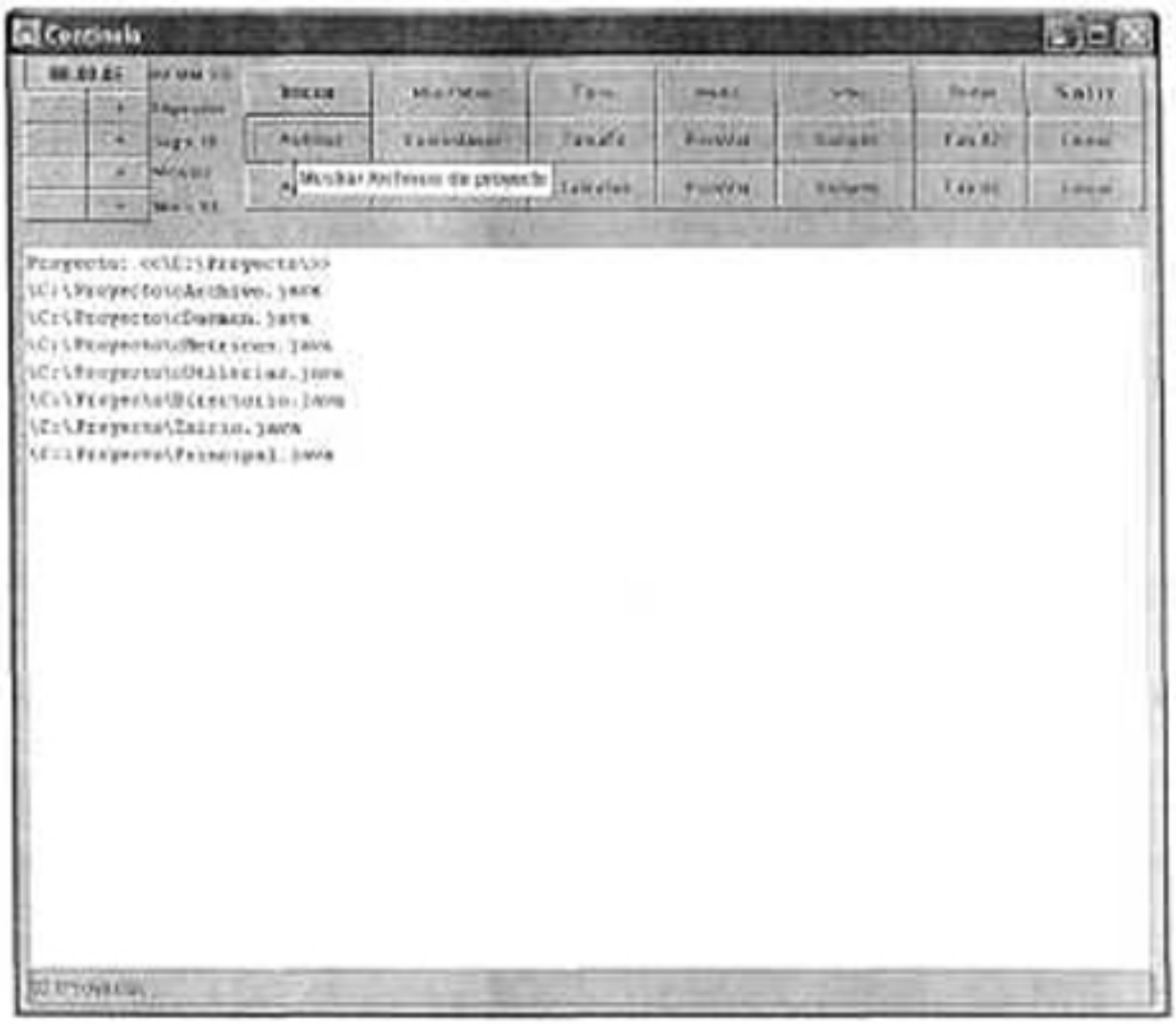

Fig. 6.3.6 Mostrar los archivos actuales

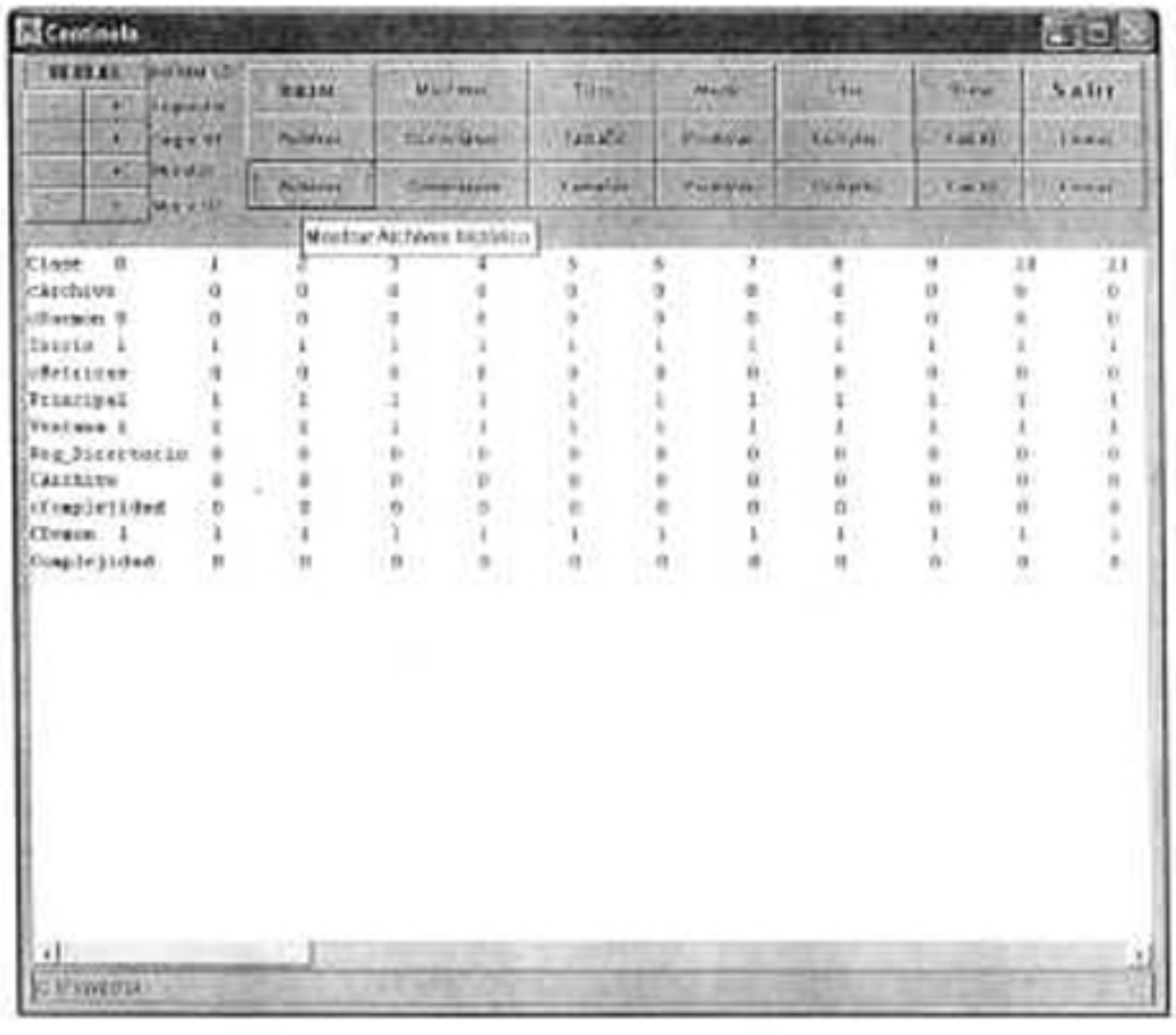

Fig. 6.3.7 Mostrar historial de archivos

En la figura 6.3.8 se presenta el porcentaje de comentarios de cada clase del proyecto actual y en la figura 6.3.9, el porcentaje de comentarios histórico.

Fig. 6.3.8 Medir el porcentaje de comentarios

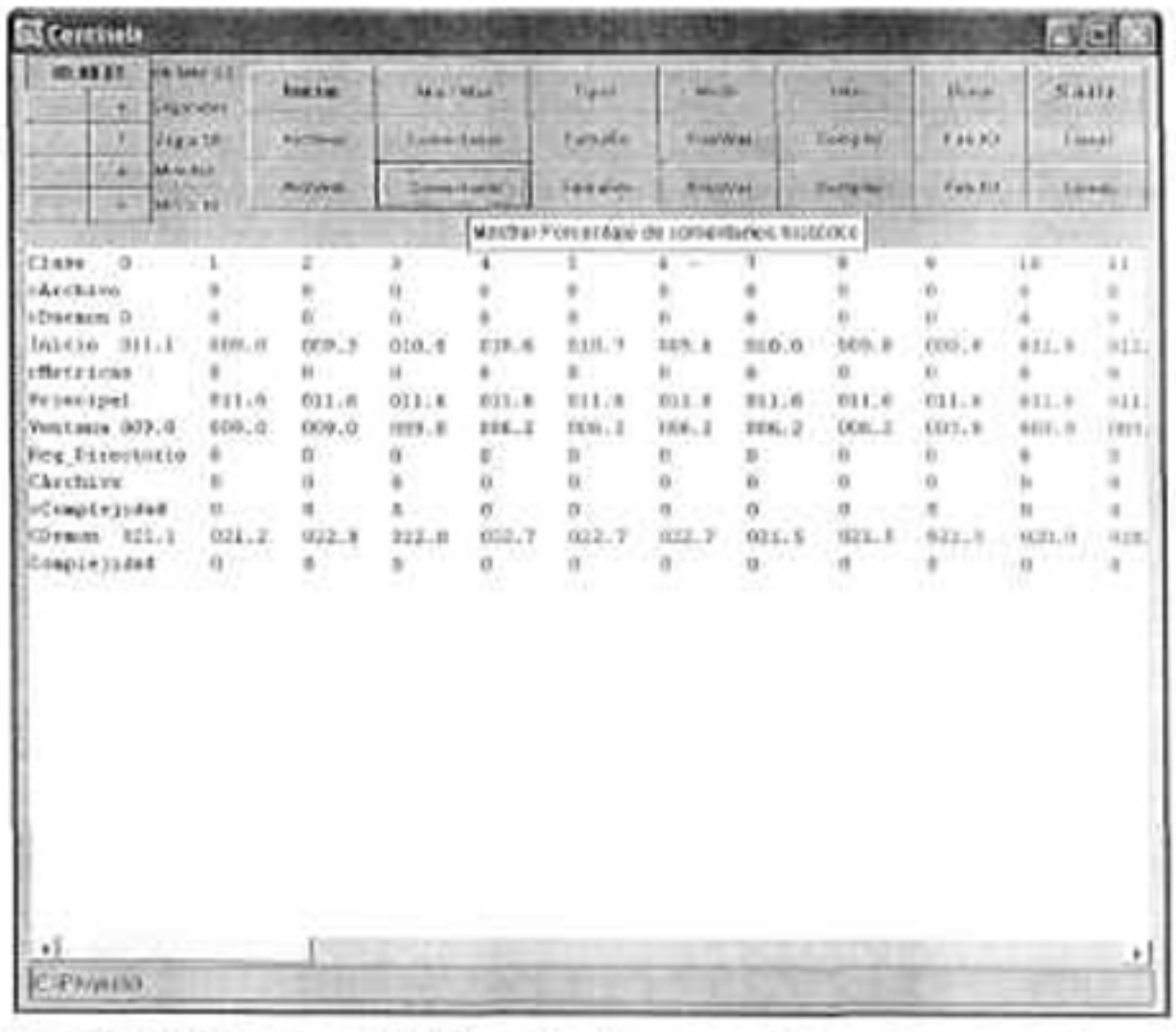

Fig. 6.3.9 Mostrar el historial de porcentajes de comentarios

Se muestra en la figura 6.3.10 el tamaño en bytes de cada clase del proyecto y en la figura 6.3.11 el tamaño de cada clase histórico.

Fig. 6.3.10 Medir el tamaño de cada clase

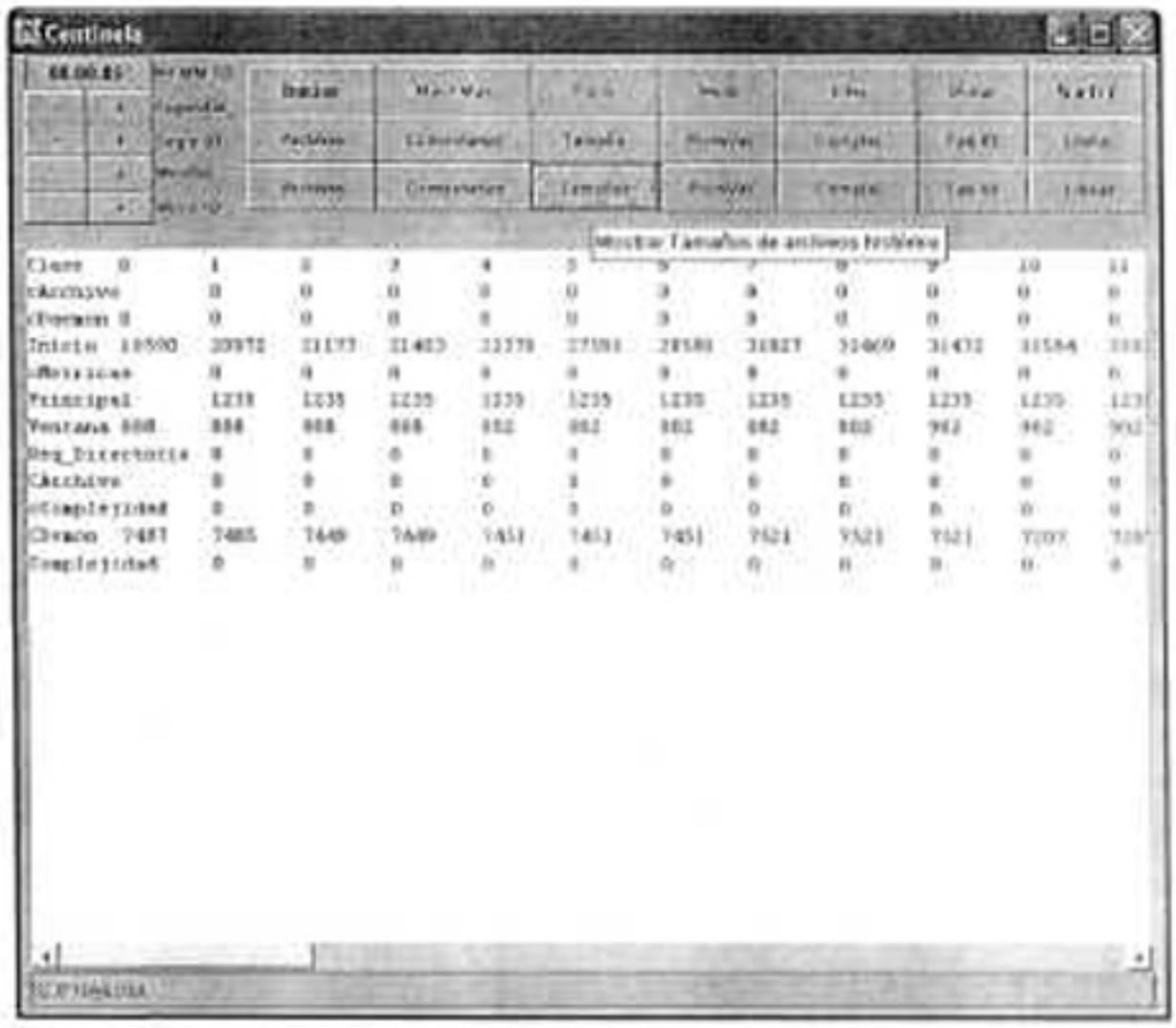

Fig. 6.3.11 Muestra el historial de tamaños de cada clase

En la figura 6.3.12 se muestra la longitud promedio de los identificadores (variables) de cada clase del proyecto y en la figura 6.3.13 el histórico.

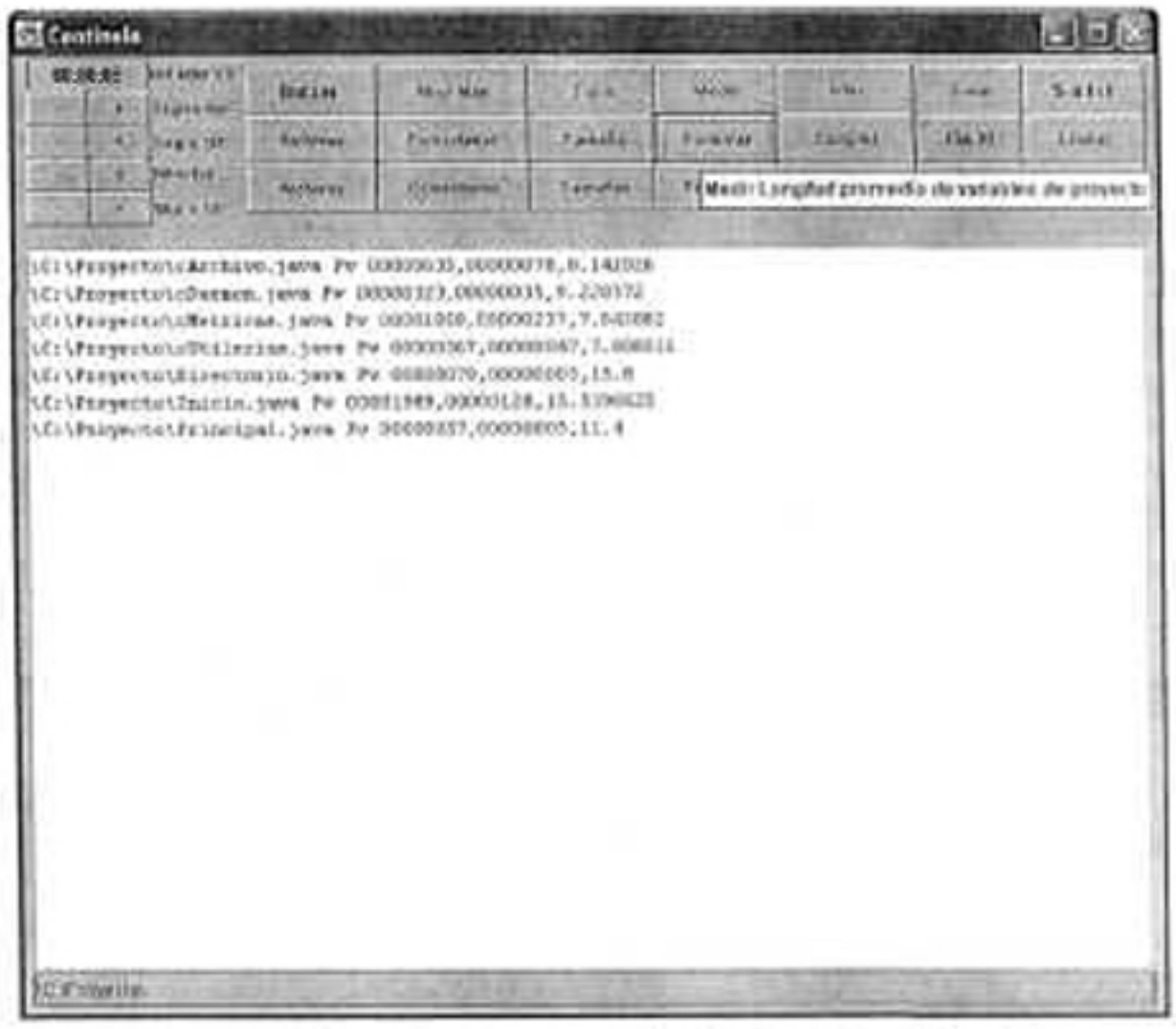

Fig. 6.3.12 Muestra la longitud promedio de los identificadores por clase

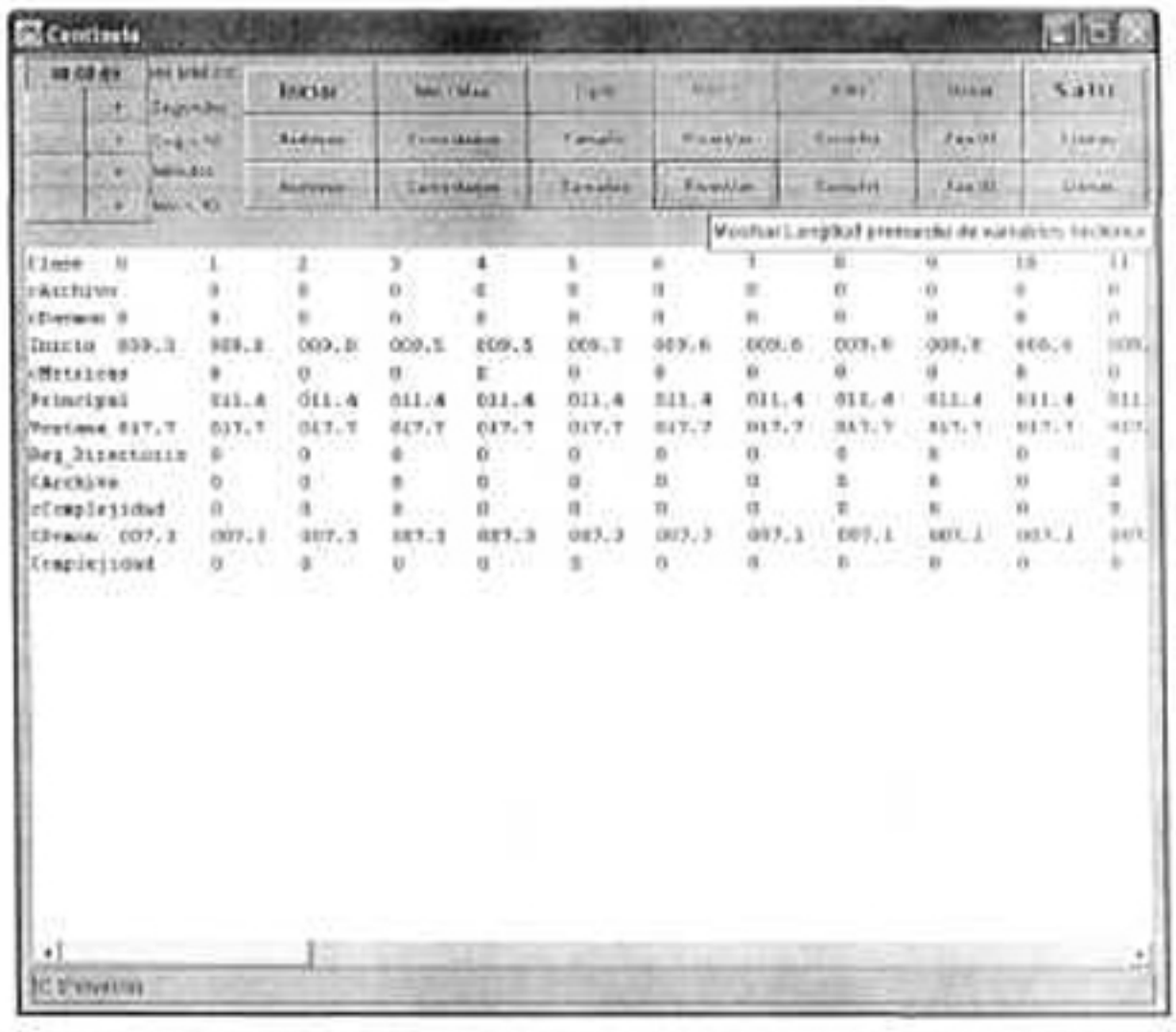

Fig. 6.3.13 Muestra el historial de la long. promedio de los identificadores por clase

En la figura 6.3.14 se muestra la complejidad ciclomática por mètodo por clase del proyecto y en la figura 6.3.15 el registro histórico de la complejidad.

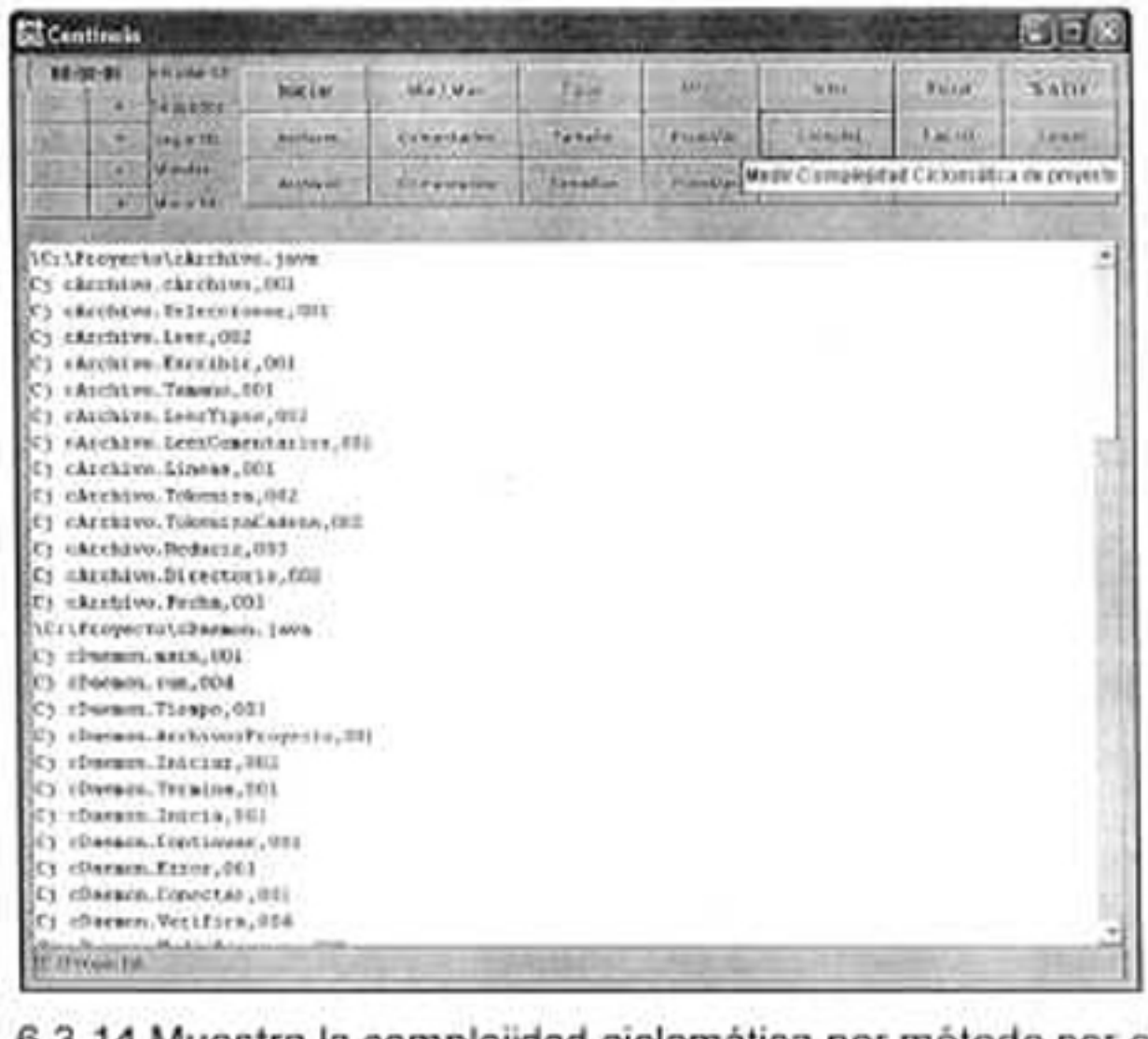

Fig. 6.3.14 Muestra la complejidad ciclomática por método por clase

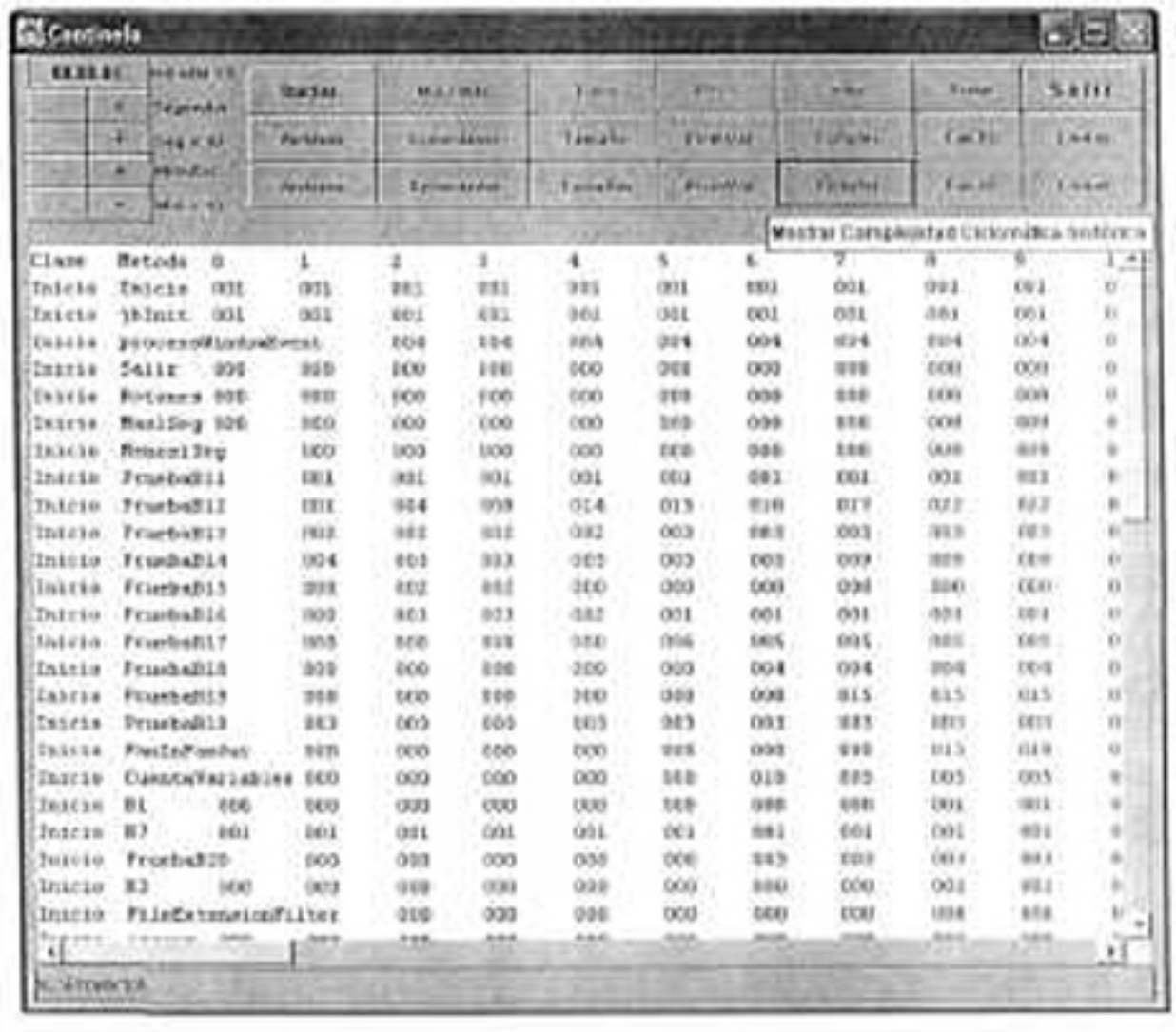

Fig. 6.3.15 Muestra el historial de la complejidad ciclomática por método por clase

En la figura 6.3.16 se muestra el *Fan In* y el *Fan Out* por método por clase del proyecto y en la figura 6.3.17 el registro histórico de *Fan In* y *Fan Out*

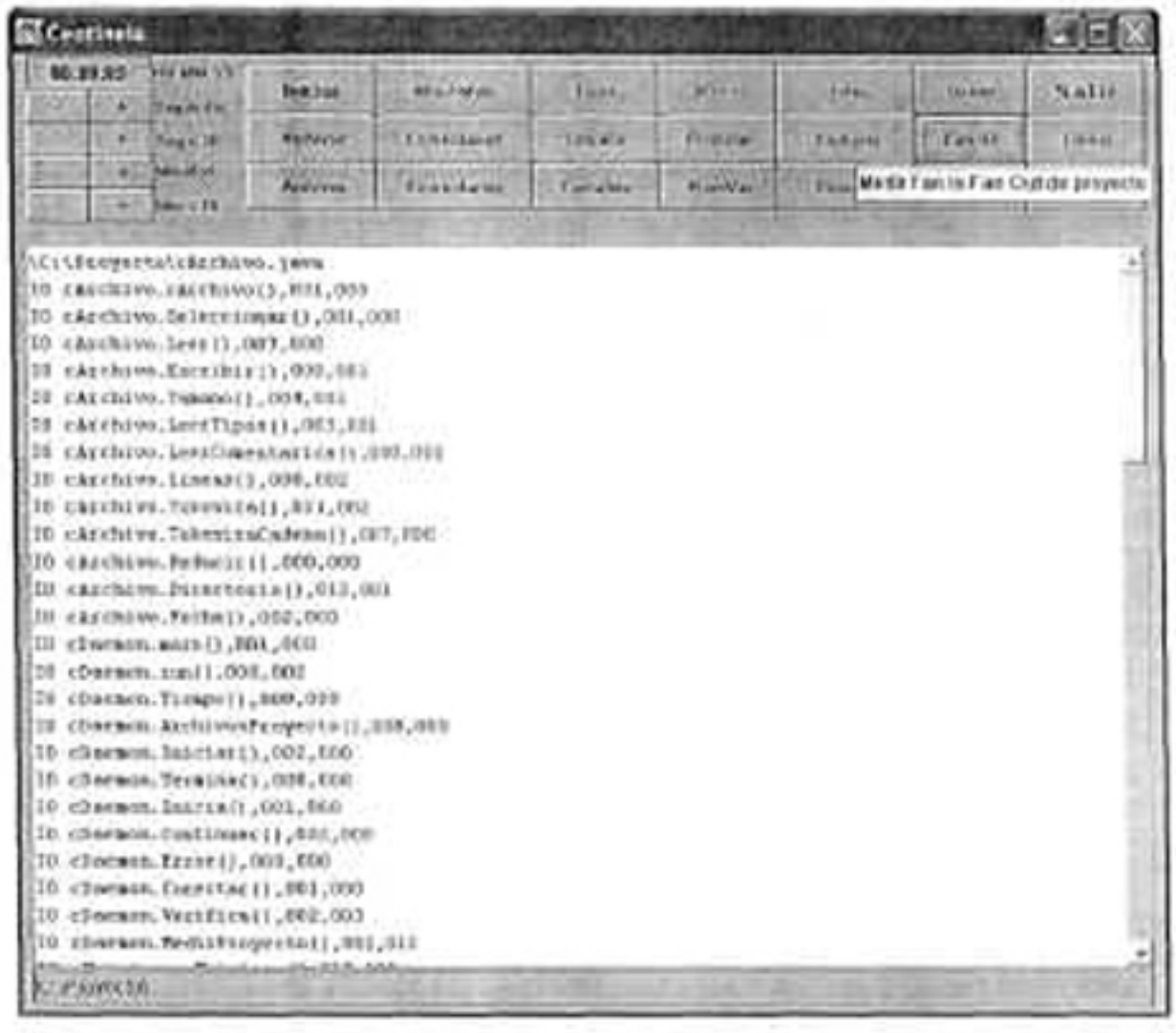

Fig. 6.3.16 Muestra el *Fan In* y el *Fan Out* por método por clase

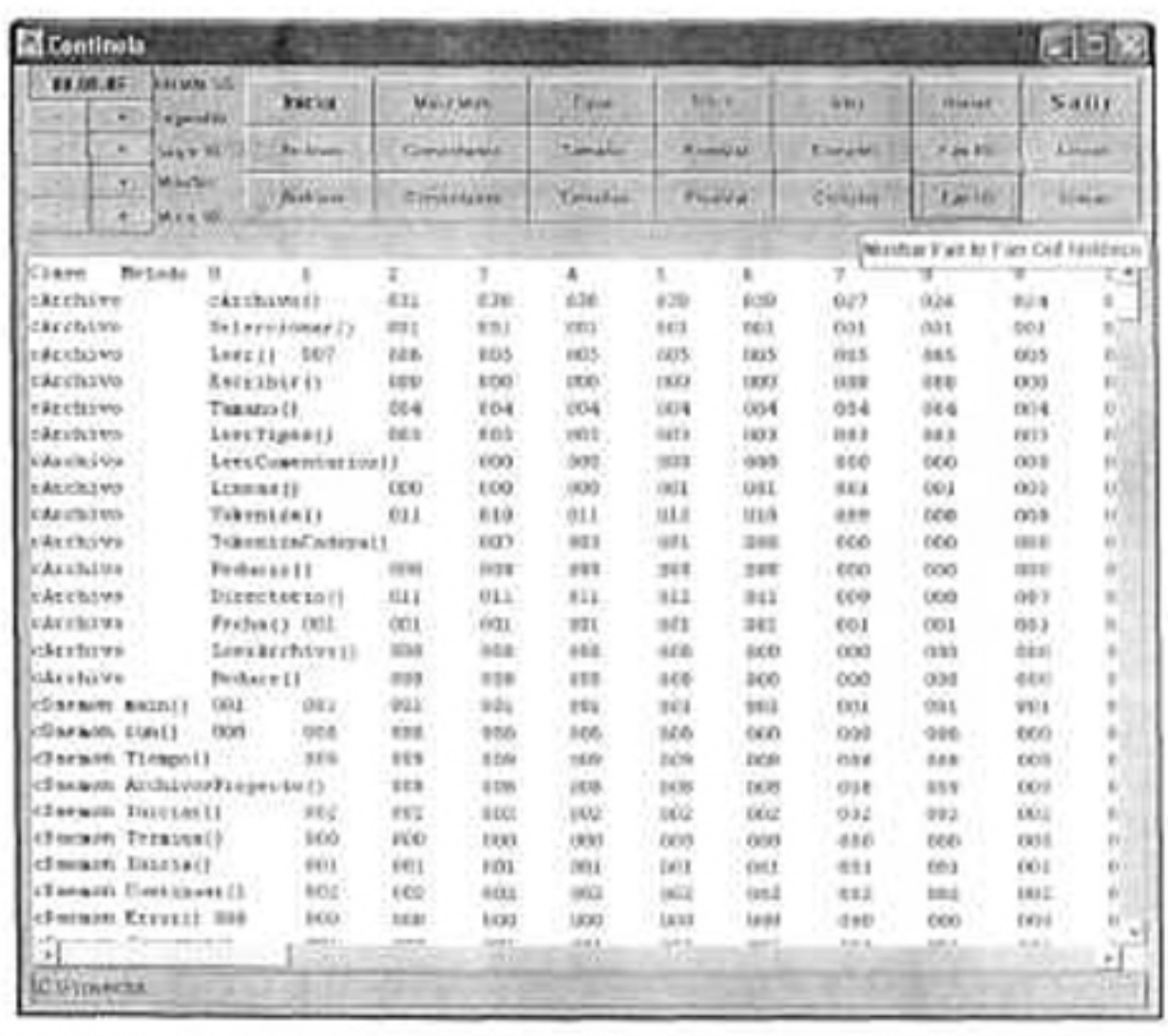

Fig. 6.3.17 Muestra el historial de *Fan In* y *Fan Out* por método por clase

En la figura 6.3.18 se muestra el número en líneas de cada clase del proyecto y en la figura 6.3.19 el registro histórico del número de líneas por clase.

Fig. 6.3.18 Muestra el número en líneas por clase

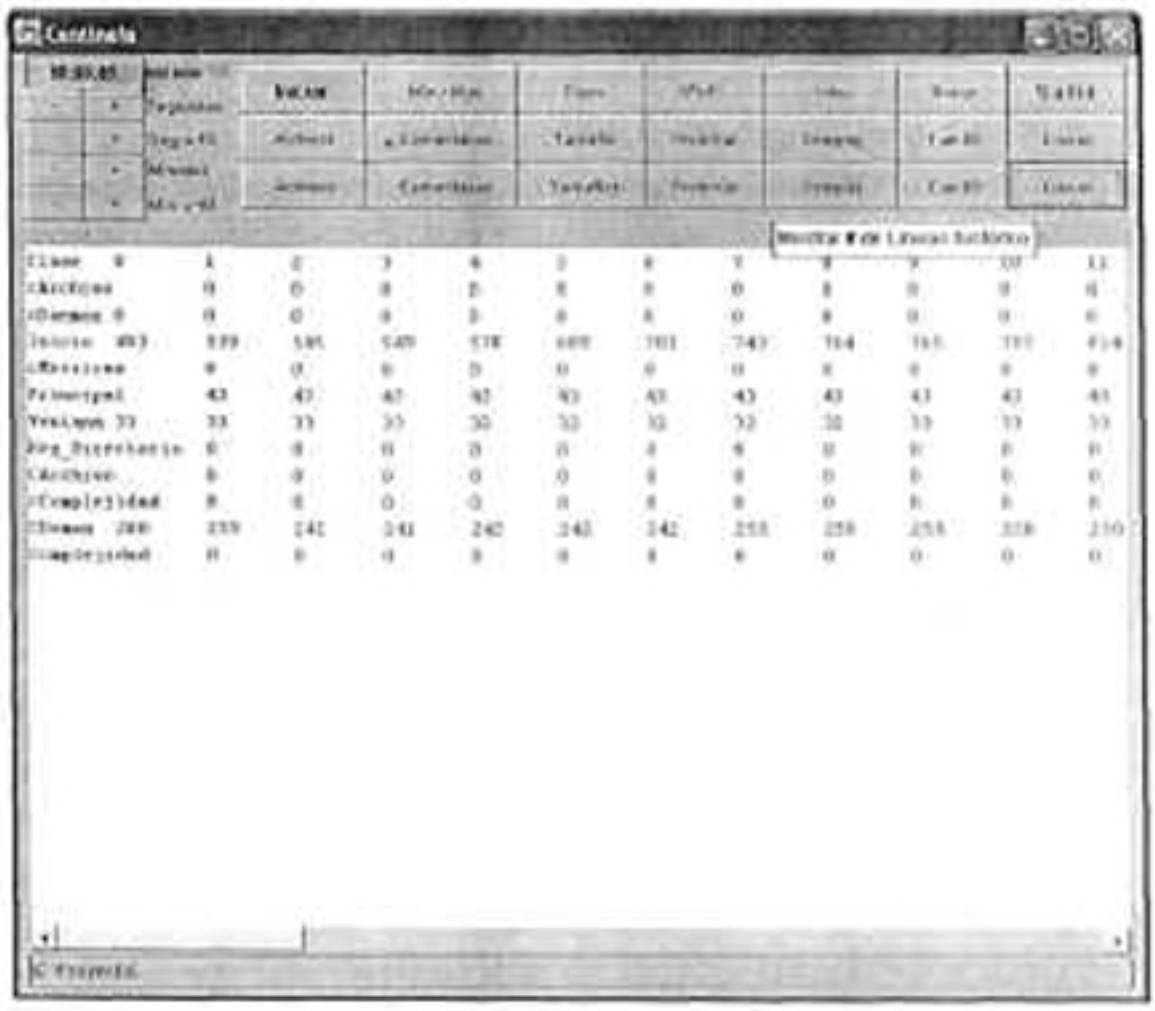

Fig. 6.3.19 Muestra el historial de números de líneas por clase

Bibliografía

1. [AIS] Apuntes de la materia Ingeniería de Software
   Pérez González, Dr. Héctor Gerardo
   Semestre Ago–Dic 2003, Posgrado de Ingeniería de la Computación, UASLP.

2. [ATSA] Apuntes de la materia Temas Selectos de Arquitectura de Software (Sistemas Multiagentes)
   Nava Muñoz, MC Sandra Edith
   Semestre Ene–Jun 2004, Posgrado de Ingeniería de la Computación, UASLP.

3. [DAS] Carmen Fernández Chamizo, Jorge Gómez Sanz, Juan Pavón Mestras
   Desarrollo de Agentes Software
   Dep. de Sistemas Informáticos y Programación
   http://grasia.fdi.ucm.es

4. [NAVA] Nava Muñoz, Sandra Edith
   Federación de Bibliotecas Digitales utilizando Agentes Móviles,
   Tesis de Maestría.
   Universidad de las Américas - Puebla, Junio de 2002.

5. [NWANA] Nwana, H.S.
   Software Agents: An Overview. Knowledge Engineering Review.
   11(3):205-244, 1996.

6. [PRESS] Pressman, Roger S.
   Ingeniería de software, 5ª Edición
   Mc Graw Hill / Interamericana de España, S.A.U., 2002

7. [SMBP] Software Metrics Best Practices PK043002.pdf
   www.visualbasic.ittoolbox.com/pub/PK043002.pdf
   Software Metrics Best Practices 2001. March 2002. Copyright© 2002, KLCI Research Group. Page 1. http://www.klci.com

8. [SOMMER] Sommerville, Ian
   Ingeniería de software, 6ª Edición
   Addison Wesley, Pearson Educación de México, 2002

9. [WEISS] Weiss, Gerhard
   Multiagent Systems, A Modern approach to Distributed Artificial Intelligence
   MIT Press, Cambridge, Massachussets, USA, 1999