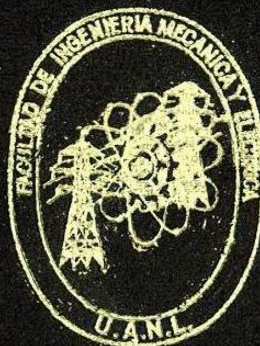
UNIVERSIDAD AUTONOMA DE NUEVO LEON

FACULTAD DE INGENIERIA MECANICA Y ELECTRICA

DIVISION DE ESTUDIOS DE POSTGRADO



METRICAS DE SOFTWARE EN LENGUAJES DE CUARTA GENERACION

TESIS

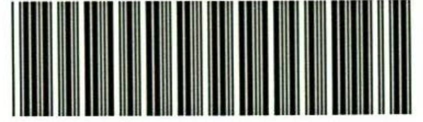
QUE PARA OBTENER EL TITULO DE
MAESTRO EN CIENCIAS DE LA ADMINISTRACION
ESPECIALIDAD EN SISTEMAS

PRESENTA

JOSE LUIS MARTINEZ FLORES

SAN NICOLAS DE LOS GARZA, N. L.
MARZO DE 1994

TM
Z5853
.M2
FIME
1994
M3


1020070680

UNIVERSIDAD AUTONOMA DE NUEVO LEON

FACULTAD DE INGENIERIA MECANICA Y ELECTRICA

DIVISION DE ESTUDIOS DE POSTGRADO

METRICAS DE SOFTWARE EN LENGUAJES DE CUARTA GENERACION

TESIS

QUE PARA OBTENER EL TITULO DE
MAESTRO EN CIENCIAS DE LA ADMINISTRACION
ESPECIALIDAD EN SISTEMAS

PRESENTA

JOSE LUIS MARTINEZ FLORES

SAN NICOLAS DE LOS GARZA, N. L.
MARZO DE 1994

UNIVERSIDAD AUTONOMA DE NUEVO LEON

FACULTAD DE INGENIERIA MECANICA Y ELECTRICA

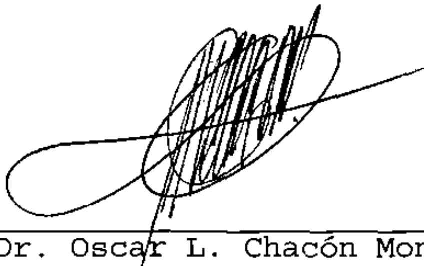
DIVISION DE ESTUDIOS DE POSTGRADO

Los miembros del comité de tesis recomendamos que la presente tesis realizada por el Lic. José Luis Martínez Flores sea aceptada como opción para obtener el grado de Maestro en Ciencias de la Administración con especialidad en Sistemas.

El comité de tesis

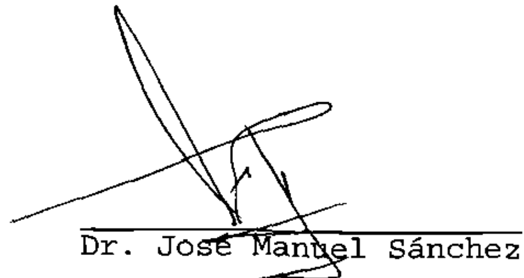
Dr. Óscar Flores Rosales

Asesor



Dr. Oscar L. Chacón Mondragón

Co-asesor



Dr. José Manuel Sánchez

Co-asesor

SAN NICOLAS DE LOS GARZA, N.L. MARZO DE 1994

TM
Z5853
.M2
FIME
1994
M3

FONDO TESIS

33088

A todas las personas que luchan por encontrarle sentido a su existencia.

AGRADECIMIENTOS

Por este medio quiero agradecer profundamente al asesor de esta tesis, Dr. Oscar Flores Rosales, por guiareme hasta la culminación de este objetivo mediante sus acertados consejos. De la misma forma, a los co-asesores Dr. Oscar L. Chacón Mondragón y Dr. José Manuel Sánchez por su valiosa colaboración.

INDICE

CONTENIDO PAGINA

INDICE i

INDICE DE TABLAS iv

INDICE DE FIGURAS vi

PROLOGO vii

RESUMEN x

CAPITULO 1.- INTRODUCCION 1

CAPITULO 2.- ANTECEDENTES.

2.0.- Introducción 4

2.1.- El Software.

2.1.1.- Concepto de Software 5

2.1.2.- Desarrollo de Software 7

2.1.3.- Problemas con el desarrollo de Software 9

2.1.4.- Causas de los problemas del desarrollo de Software 9

2.2.- La Ingeniería de Software.

2.2.1.- Definición 11

2.2.2.- Paradigmas de la Ingeniería del Software 13

2.2.3.- Unificación de los paradigmas.. 20

2.3.- Métricas de Software 23

2.4.- Métricas de Calidad del Software.

2.4.1.- Calidad del Software 26

2.4.2.- Factores de calidad del Software 27

2.4.3.- Métricas Cualitativas de la calidad del Software 29

2.4.4.- Métricas Cuantitativas de la calidad del Software 31

2.5.- Resúmen 34

CAPITULO 3.- METRICAS DE HALSTEAD Y PREGUNTAS DE LA INVESTIGACION.

3.0.- Introducción 35

3.1.- Importancia 36

3.2.- Definiciones Básicas 36

3.3.- Estimador de la longitud de un programa 40

3.4.- Estimador del volúmen potencial de un programa 41

3.5.- Estimador del nivel/dificultad de un programa y definición del contenido de inteligencia de un programa 42

3.6.- Estimador del esfuerzo y del tiempo de programación de un programa 43

3.7.- Definición del nivel del lenguaje de programación y su estimador 45

3.8.- Propuesta de preguntas e hipótesis de investigación 47

3.9.- Resúmen 48

CAPITULO 4.- METODOLOGIA DE LA INVESTIGACION.

4.0.- Introducción 49

4.1.- Diseño de investigación 49

4.2.- Selección de los 4GL's 50

4.3.- Selección de la muestra 50

4.4.- Desarrollo y validación del analizador de código 51

4.5.- Resúmen 52

CAPITULO 5.- ANALISIS DE DATOS.

5.0.- Introducción 53

5.1.- Revisión de la exactitud de los datos 53

5.2.- Estadísticas Descriptivas 55

5.3.- Análisis de datos de la primera hipótesis de investigación 59

5.4.- Análisis de datos de la segunda hipótesis de investigación 61

5.5.- Resúmen 65

CAPITULO 6.- CONCLUSIONES 67

APENDICE A.- ANALIZADOR DE CODIGO FUENTE........ 71

BIBLIOGRAFIA 78

LISTA DE TABLAS

TABLA PAGINA

Tabla 2.1: Factores y métricas de calidad................ 31

Tabla 3.1: Media y varianza del nivel del lenguje para seis lenguajes........................... 46

Tabla 3.2: Distribuciones de frecuencia de niveles del lenguaje para diferentes lenguajes........ 47

Tabla 5.1: Longitudes reales y estimadas para DBASE III.................................. 55

Tabla 5.2: Longitudes reales y estimadas para FOXPRO2.................................... 56

Tabla 5.3: Niveles del lenguaje para DBASE III........... 57

Tabla 5.4: Niveles del lenguaje para FOXPRO2............. 57

Tabla 5.5: Media y desviación estándar para los niveles del lenguaje DBASE III y FOXPRO2...... 58

Tabla 5.6: Tabla de frecuencias para el nivel del lenguaje DBASE III........................... 58

Tabla 5.7: Tabla de frecuencias para el nivel del lenguaje FOXPRO2............................. 59

Tabla 5.8: Coeficientes de correlación de Pearson entre N y Nest............................. 60

Tabla 5.9: Resultados de las pruebas Z entre los 4GL's y los 3GL's.......................... 63

Tabla 5.10: Resultados de la prueba Z, entre el nivel del lenguaje de los 4GL's y el Inglés Prosaico.................................. 64

TABLA PAGINA

Tabla 5.11: Resultados de la prueba Z, comparando las medias de los niveles del lenguaje para FOXPRO2 y DBASE III........................... 65

LISTA DE FIGURAS

FIGURA PAGINA

Fig. 2.1: Curva de fallas del hardware.................. 6

Fig. 2.2: Curva de fallas del software.................. 6

Fig. 2.3: El ciclo de vida clásico...................... 15

Fig. 2.4: Creación de prototipos........................ 17

Fig. 2.5: El modelo en espiral.......................... 18

Fig. 2.6: Clasificación de métricas..................... 25

Fig. 2.7: Factores de calidad del software de McCall..... 28

Fig. 2.8: Complejidad de la red de flujo de control...... 33

Fig. 6.1: Relación ideal entre la longitud real y la longitud estimada de un programa, y correlación lineal práctica.................. 68

Fig. 6.2: Relación ideal entre el nivel del lenguaje y la longitud real de un programa............... 70

Fig. A.1: Diagrama de contexto del analizador de código.................................. 77

Fig. A.2: Diagrama de flujo de datos del analizador de código.................................. 77

PROLOGO

Al menos en nuestro entorno regional es muy común que algunos analistas de sistemas hagan estimaciones de software en base a su experiencia. Aunque ésto puede dar buenos resultados si el analista tiene mucha experiencia, deja mucho que desear la formalidad con que se hacen las estimaciones de software.

Otro hecho, es que a algunos analistas solamente les interesa que el software "funcione". Cuando el software está funcionando, generalmente se olvidan de él para empezar a desarrollar otro. Esto puede causar problemas cuando el software requiera mantenimiento, manifestando por lo tanto, la baja calidad del software.

Dado lo anterior, sería conveniente encontrar alguna forma de medir la calidad del software que se desarrolla, así como estimar su tamaño y tiempo de desarrollo.

Por otra parte, es un hecho que los lenguajes de cuarta generación (4GL's), desde los pioneros como el DBASE en todas sus versiones y continuando con el FOXPRO, CLIPPER, FOXBASE, PARADOX, hasta los actualmente más poderosos 4GL's de los 90's como el PROGRESS y el ORACLE, cada vez son más utilizados para el desarrollo de software.

Por consiguiente, es muy recomendable el análisis de código de los 4GL's, para encontrar una forma de estimar el tamaño y tiempo de desarrollo del software, así como para encontrar una forma de medir la calidad del software que se desarrolla.

La literatura nos indica que una forma de encontrar estimaciones del tamaño y tiempo de desarrollo del software, así como formas de medir la calidad del software que se desarrolla, es por medio del código fuente de los programas que forman el software. Estas estimaciones son válidas para lenguajes de tercera generación y no se ha demostrado su validez para lenguajes de cuarta generación.

La presente tesis da una solución parcial a este problema, analizando: 1) los 4GL's DBASE III, FOXPRO2; 2) El estimador de la longitud de un programa escrito en DBASE III o FOXPRO2; 3) Los niveles de los lenguajes DBASE III y FOXPRO2.

Adicionalmente, se anexa a la presente tesis una herramienta que puede ayudar a los analistas de sistemas a hacer estimaciones de software implementado en FOXPRO2 o en DBASE III. Esta herramienta la llamamos: analizador de código. En el apéndice A se da una breve descripción de su funcionamiento.

El analizador hace un análisis de código fuente escrito en FOXPRO2 o en DBASE III, pero con algunas modificaciones puede hacer análisis de código fuente escrito en cualquier lenguaje.

Espero que esta tesis pueda dar algunas guías para hacer mejores estimaciones y revisiones de la calidad del software.

JOSE LUIS MARTINES FLORES.

MARZO DE 1994.

RESUMEN

Las métricas de Halstead de las Ciencias del Software son de mucha utilidad para que los administradores de proyectos de software hagan estimaciones del tamaño y del tiempo de desarrollo de éstos. Los resultados de esta teoría se basan en muestras de programas escritos en lenguajes máquina, ensamblador y de tercera generación.

Halstead propone un estimador de la longitud de un programa. Se ha demostrado empíricamente que este estimador es válido para lenguajes de tercera generación. Halstead define el nivel de un lenguaje y hace una clasificación de diferentes lenguajes. Esta clasificación tiene sentido práctico.

La tesis tiene por objetivo responder las siguientes preguntas de investigación:

1) ¿Es el estimador de longitud propuesto por Halstead un buen estimador para un programa escrito en algún 4GL?

2) ¿Es el nivel del lenguaje de los 4GL's mayor que el nivel del lenguaje de los 3GL's y menor que el nivel del lenguaje natural?

Para responder estas preguntas se procedió de la siguiente manera:

1) Se seleccionaron los 4GL's para hacer la inevestigación (DBASE III, FOXPRO2).

2) Se seleccionó la muestra para los lenguajes antes mencionados (26 elementos para DBASE III, 117 elementos para FOXPRO2).

3) Se construyó un analizador de código fuente para obtener las métricas de Halstead, para cada uno de los programas.

4) Se hizo un análisis de correlación para responder la primera pregunta de investigación.

5) Se hicieron pruebas Z para responder la segunda pregunta de investigación.

Mediante un análisis de los datos obtenidos se llegó a las siguientes conclusiones:

1) El estimador de la longitud de un programa propuesto por Halstead es un buen estimador en los lenguajes FOXPRO2 y DBASE III.

2) Hay evidencia estadística de que el nivel del lenguaje FOXPRO2 es mayor que el nivel del lenguaje de los 3GL's. Con el lenguaje DBASE III no hubo suficiente evidencia estadística pero sí numérica.

3) No hay evidencia estadística de que los niveles de los lenguajes FOXPRO2 y DBASE III sean diferentes.

4) No hay evidencia estadística de que los niveles de los lenguajes FOXPRO2 y DBASE III sean menores que el nivel del lenguaje natural.

CAPITULO I

INTRODUCCION

Hay muchos factores que afectan el desarrollo de un programa. Estos incluyen: el tipo de programa que está siendo desarrollado, el tamaño del programa, el lenguaje de implementación, la experiencia de los programadores involucrados, las técnicas de programación involucradas, y el ambiente computacional.

La ciencia del software (Halstead, 1977) es un modelo del proceso de programación que se basa en un número manipulable de los principales factores que afectan la programación. Esto ofrece una guía hacia estimadores que pueden ser útiles a los administradores de proyectos de software.

Los resultados de esta teoría se han utilizado y se siguen utilizando (Wrigley, 1991; Bowman, 1990), con sus correspondientes críticas y evaluaciones (Shen, 1983; Weyucker, 1988; Ramamurthy, 1988).

Los resultados de esta teoría se basan en muestras de programas escritos en lenguaje máquina, ensamblador y de tercera generación. Por lo tanto, cabe cuestionarse lo

siguiente: ¿Los resultados de las ciencias del software tienen sentido en los lenguajes de cuarta generación (4GL's)?

La pregunta es bastante compleja. Sin embargo pudiéramos contestarla analizando un estimador de los que propone Halstead, y clasificando algunos 4GL's dentro de la tabla de clasificaciones de Halstead.

Halstead propone un estimador de la longitud de un programa (Nest). Hay evidencia empírica que demuestra que Nest es un buen estimador para lenguajes de tercera generación (3GL's) (Shen, 1983), pero ¿Es Nest un buen estimador de la longitud de un programa para los 4GL's?

Halstead, además hace una clasificación de diferentes niveles de lenguaje (Inglés Prosaico, PL/I, Algol 58, Fortran, Pilot, Ensamblador). ¿Tiene sentido clasificar los 4GL's, como lo hizo Halstead? Esto es, ¿Es el nivel del lenguaje de los 4GL's mayor que el nivel del lenguaje de los 3GL's y menor que el nivel del lenguaje Inglés Prosaico (lenguaje natural)?

La presente tesis trata de dar respuesta a las anteriores preguntas mediante el logro de los siguientes objetivos:

Objetivo 1. Determinar si el estimador de la longitud de un programa, como el propuesto por Halstead (1977), es un buen estimador para lenguajes de cuarta generación.

Objetivo 2. Determinar si el nivel del lenguaje de los 4GL's es mayor que el nivel del lenguaje de los 3GL's y menor que el nivel del lenguaje natural (Inglés Prosaico).

CAPITULO 2

ANTECEDENTES

2.0 INTRODUCCION

El objetivo fundamental de la investigación es hacer un análisis de las métricas de Halstead en lenguajes de cuarta generación (4GL's). Por lo tanto, es importante dar una explicación del lugar en que se ubican las métricas de Halstead, así como el de los conceptos que nos conducen a su significado.

El presente capítulo se conducirá de la siguiente forma. En 2.1 se dará una explicación del concepto de software, de su desarrollo y de sus problemas. En 2.2 se dará una explicación de lo que es la ingeniería de software, de tres diferentes conceptualizaciones, y, finalmente, de la unificación de tales conceptualizaciones. En 2.3 se dará una explicación de lo que son las métricas de software, su clasificación y sus dimensiones. En 2.4 se dará una explicación de las métricas de calidad del software, y su clasificación.

2.1 EL SOFTWARE

2.1.1 CONCEPTO DE SOFTWARE.

El software es un concepto muy difícil de definir, aunque se le diera una definición formal se podrían encontrar otras definiciones más completas. Se necesita algo más que una definición formal para poder comprender lo que es el software. Sin embargo, podemos definir software como: "programas o conjunto de instrucciones que dirigen las operaciones de procesamiento de información ejecutadas por medio del hardware" (Athey, 1988, p. 15).

Para comprender lo que es el software, es importante examinar sus características que lo hacen ser diferente de otras cosas que el hombre puede construir, como el hardware. El software es un elemento del sistema que es lógico, en lugar de físico, por lo tanto:

1. El software se desarrolla, no se fabrica en un sentido clásico.

2. El software no se estropea con el tiempo. (Ver Fig. 2.1 y Fig. 2.2).

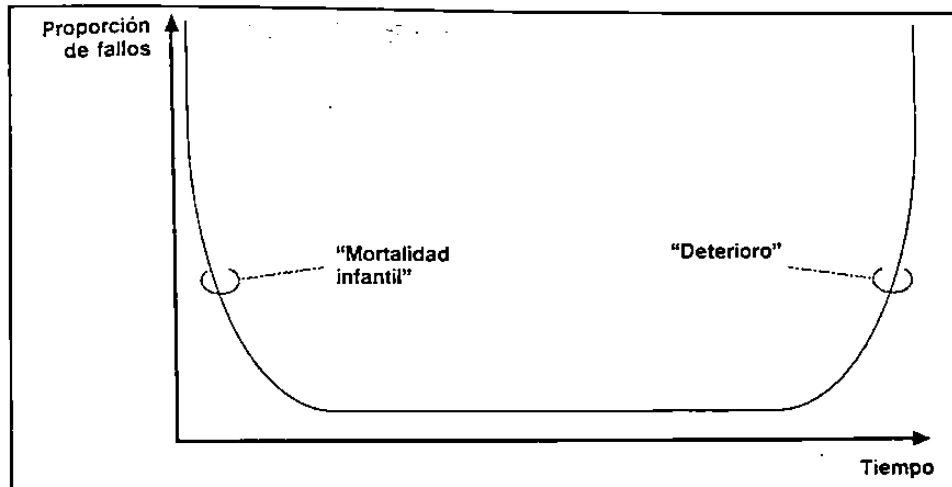


Fig. 2.1 Curva de fallas del hardware.

Fuente: (Pressman, 1993).

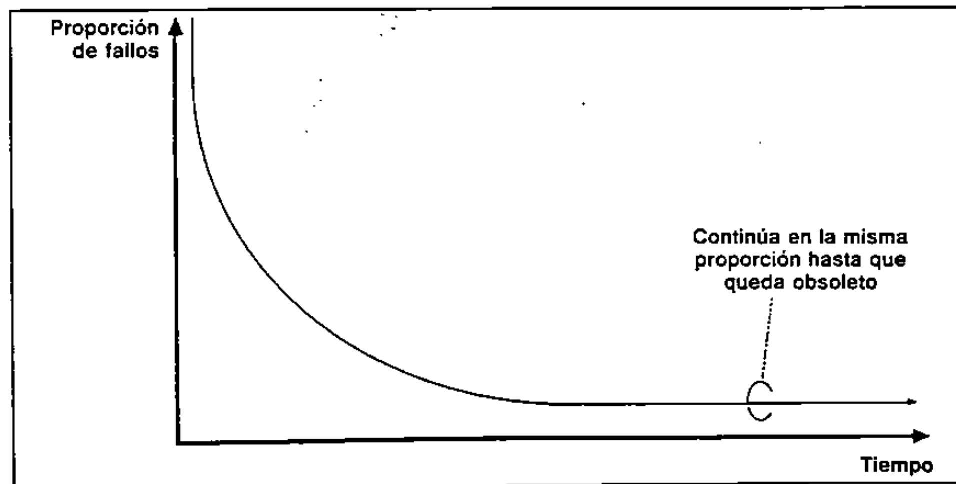


Fig. 2.2 Curva de fallas del software (idealizada).
Fuente: (Pressman, 1993).

3. La mayoría del software se construye a la medida, en vez de ensamblar componentes existentes. Esta caracteristica tiende a desaparecer. La reusabilidad del software va ser, en la presente década de los 90's, uno de los principales

contribuidores en la productividad y calidad del software (Yourdon, 1992).

2.1.2 DESARROLLO DE SOFTWARE.

El software se desarrolla mediante un lenguaje de programación que tiene un vocabulario limitado, una gramática definida explícitamente y reglas bien formadas de sintaxis y semántica. Las clases de lenguajes que se utilizan actualmente son los lenguajes máquina, los lenguajes de alto nivel y los lenguajes no procedimentales (más adelante se explica lo que es un lenguaje procedimental y no procedimental)

Los lenguajes máquina son una representación simbólica del conjunto de instrucciones de la unidad central de proceso (CPU). Si un buen programador produce programas mantenibles y bien documentados, puede utilizarse el lenguaje máquina para hacer un uso extremadamente eficiente de la memoria y optimizar la velocidad de ejecución del programa.

Los lenguajes de alto nivel (por ejemplo, COBOL, FORTRAN, PASCAL, C, ADA, C^{++}, OBJECT PASCAL, APL, LISP, etc.) permiten al programador y al programa independizarse de la máquina. Cuando se utiliza un traductor sofisticado,

el vocabulario, la gramática, la sintaxis y la semántica de un lenguaje de alto nivel pueden ser mucho más sofisticados que los lenguajes máquina. Los compiladores e intérpretes de los lenguajes de alto nivel producen lenguaje máquina como salida.

El código máquina, los lenguajes ensambladores (nivel máquina) y los lenguajes de programación de alto nivel son normalmente considerados como las tres primeras generaciones de lenguajes de computadoras. Con cualquiera de estos lenguajes, el programador se preocupa tanto de la especificación de la estructura de la información, como de la especificación del control del propio programa. Debido a ésto, los lenguajes de las tres primeras generaciones se denominan lenguajes procedimentales.

En los 80's surgió un grupo de lenguajes no procedimentales, también llamados de cuarta generación, por ejemplo DBASE, FOXPRO, FOXBASE, PARADOX; y actualmente, en los 90's, están surgiendo poderosos lenguajes de cuarta generación como PROGRESS y ORACLE. En vez de requerir que quien desarrolla el software especifique los detalles procedimentales, un programa en un lenguaje no procedimental hace la "especificación del resultado deseado, en vez de la especificación de la acción requerida para conseguir el resultado" (Pressman, 1993). Hasta la fecha, los lenguajes

de cuarta generación se han utilizado en aplicaciones de base de datos.

2.1.3 PROBLEMAS CON EL DESARROLLO DE SOFTWARE.

Los problemas que afligen al desarrollo de software se pueden caracterizar bajo perspectivas diferentes, pero se centran en las siguientes (Pressman, 1993):

1) La planificación y estimación de costos son frecuentemente imprecisas.
2) La productividad de los desarrolladores de software no satisface lo que los usuarios demandan.
3) La calidad del software es cuestionable.

2.1.4 CAUSAS DE LOS PROBLEMAS DEL DESARROLLO DE SOFTWARE.

Los problemas con el desarrollo de software se han producido por la propia naturaleza del software y por los errores de las personas encargadas de su desarrollo. Como se dijo anteriormente, el software es un elemento lógico en vez de físico y por lo tanto, el éxito se mide por la calidad de él como un todo.

El software no se rompe. Si se encuentran fallas, lo más probable es que se introdujeran inadvertidamente durante el desarrollo y no se detectaran durante la prueba.

2.2 LA INGENIERIA DE SOFTWARE

2.2.1 DEFINICION.

No existe un enfoque que sea el mejor para la solución de los problemas del software. Sin embargo, mediante la combinación de métodos completos para todas las fases del desarrollo de software, mejores herramientas para automatizar estos métodos, bloques de construcción más potentes para la implementación de software, mejores técnicas para la garantía de calidad del software y una filosofía predominante para la coordinación, control y administración, podemos encontrar una disciplina para el desarrollo de software -ingeniería de software.

Una definición de la ingeniería de software (Sigwart, 1990, p. 6) es la siguiente:

"La ingeniería de software es el enfoque sistemático hacia la especificación, desarrollo, operación, mantenimiento y retiro del software."

Como se comentó en el concepto de software, se necesita comprender lo que es la ingeniería de software más que su definición formal. La ingeniería de software abarca un conjunto de tres elementos clave: métodos, herramientas y

Reemplazamos las partes defectuosas durante el mantenimiento del software; es decir, el mantenimiento incluye normalmente la corrección o modificación del diseño.

Frecuentemente, los responsables del desarrollo de software han sido ejecutivos de nivel medio y alto, sin conocimientos de software. El administrador de software debe comunicarse con todas las partes implicadas en el desarrollo del software -clientes, desarrolladores, equipo de soporte, y otros. La comunicación puede romperse debido a que se comprenden mal las características especiales del software y los problemas particulares asociados con su desarrollo.

Los programadores (ingenieros de software) han tenido muy poco entrenamiento formal en las nuevas técnicas del desarrollo de software. Cada individuo enfoca su tarea de programación con la experiencia obtenida en trabajos anteriores. Algunas personas desarrollan un método ordenado y eficiente del desarrollo de software mediante prueba y error, pero muchos otros desarrollan malos hábitos que dan como resultado una pobre calidad y mantenibilidad del software (mantenibilidad del software significa la facilidad con que se le puede dar mantenimiento al software).

2.2 LA INGENIERIA DE SOFTWARE

2.2.1 DEFINICION.

No existe un enfoque que sea el mejor para la solución de los problemas del software. Sin embargo, mediante la combinación de métodos completos para todas las fases del desarrollo de software, mejores herramientas para automatizar estos métodos, bloques de construcción más potentes para la implementación de software, mejores técnicas para la garantía de calidad del software y una filosofía predominante para la coordinación, control y administración, podemos encontrar una disciplina para el desarrollo de software -ingeniería de software.

Una definición de la ingeniería de software (Sigwart, 1990, p. 6) es la siguiente:

"La ingeniería de software es el enfoque sistemático hacia la especificación, desarrollo, operación, mantenimiento y retiro del software."

Como se comentó en el concepto de software, se necesita comprender lo que es la ingeniería de software más que su definición formal. La ingeniería de software abarca un conjunto de tres elementos clave: métodos, herramientas y

procedimientos. Estos elementos ayudan al administrador a controlar el proceso de desarrollo de software y a suministrar, a los que practican dicha ingeniería, las bases para construir software de alta calidad de una forma productiva.

Los métodos de la ingeniería de software indican "cómo" construir técnicamente el software. Los métodos abarcan un amplio espectro de tareas que incluyen: planificación y estimación de proyectos, análisis de los requisitos del sistema y del software, diseño de estructura de datos, arquitectura de programas y procedimientos algorítmicos, codificación, prueba y mantenimiento.

Las herramientas de la ingeniería de software suministran un soporte automático o semiautomático para los métodos. Cuando se integran las herramientas de tal manera que la información creada por una herramienta pueda ser usada por otra, se establece un sistema para el soporte del desarrollo de software llamado ingeniería de software asistida por computadora (CASE).

Los procedimientos de la ingeniería de software son el aglutinante de los métodos y herramientas, que facilita un desarrollo racional y oportuno del software. Los procedimientos definen la secuencia en la que se aplican los

métodos, las entregas (documentos, reportes, formas, etc.) que se requieren, los controles que ayudan a asegurar la calidad y coordinar los cambios, y las directrices que ayudan a los administradores de software a evaluar el progreso.

2.2.2 PARADIGMAS DE LA INGENIERIA DE SOFTWARE.

La ingeniería de software está compuesta por una serie de pasos que abarcan los métodos, las herramientas y los procedimientos. Estos pasos se denominan frecuentemente paradigmas de la ingeniería de software. La elección de un paradigma para la ingeniería de software se lleva a cabo de acuerdo con la naturaleza del proyecto y de la aplicación. Existen tres paradigmas clásicos que se han debatido ampliamente: el ciclo de vida clásico, la construcción de prototipos, y el modelo en espiral.

El ciclo de vida clásico.

El paradigma del ciclo de vida clásico (Fig. 2.3) para la ingeniería de software exige un enfoque sistemático y secuencial del desarrollo de software. Este paradigma abarca las siguientes actividades:

a) Ingeniería y análisis del sistema. En esta actividad primero se establecen los requisitos de todos los elementos

del sistema y luego se asigna algún subconjunto de estos requisitos al software. Este planteamiento del sistema es esencial cuando el software debe interrelacionarse con otros elementos, tales como hardware, personas y bases de datos.

b) Análisis de los requisitos del software. El proceso de recopilación de los requisitos se centra e intensifica especialmente para el software.

c) Diseño. El proceso de diseño traduce los requisitos en una representación de software que obtenga la calidad requerida antes de que comience la codificación.

d) Codificación. En el proceso de codificación el diseño se traduce en una forma legible para la máquina.

e) Prueba. Una vez que se ha generado el código, comienza la prueba del programa. La prueba se centra en la lógica interna del software, asegurando que todos los estatutos se han probado; y para las funciones externas, realizando pruebas que aseguren que la entrada ya definida produzca los resultados que realmente se requieren.

f) Mantenimiento. Indudablemente, el software sufrirá cambios después de que se entregue al cliente. Los cambios ocurrirán debido a que se hayan encontrado errores, a que el software deba adaptarse a cambios de entorno externo (por ejemplo, un cambio solicitado debido a que se tiene un nuevo requisito de la aplicación, sistema operativo o dispositivo periférico), o debido a que el cliente requiera ampliaciones funcionales o del rendimiento. El mantenimiento del software

aplica cada uno de los pasos precedentes del ciclo de vida a un programa existente en vez de a uno nuevo.

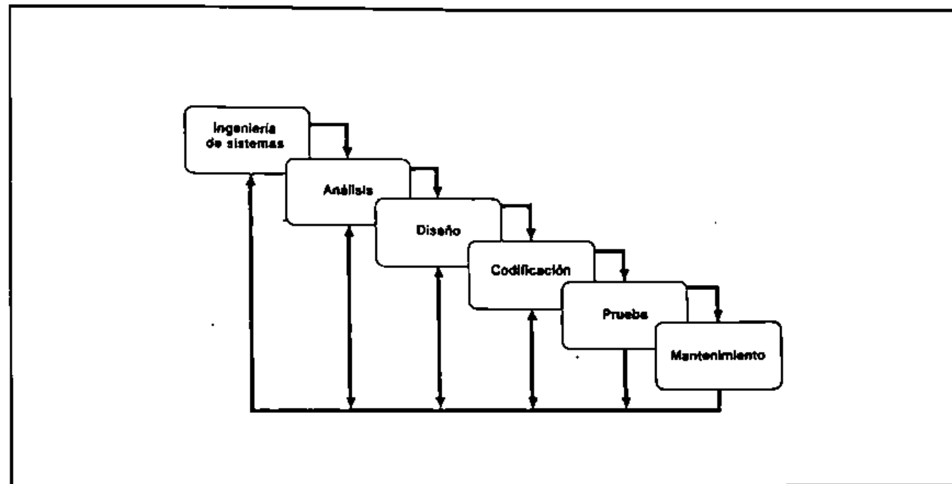


Fig. 2.3 El ciclo de vida clásico.
Fuente: (Pressman, 1993)

Construcción de prototipos.

La construcción de prototipos es un proceso que facilita al programador la creación de un modelo de software a construir. El modelo tomará una de las tres formas siguientes:

a) Un prototipo en papel o un modelo basado en computadora que describa la interacción hombre-máquina, de forma que facilite al usuario la comprensión de cómo se producirá tal interacción.

b) Un prototipo que implemente algunos subconjuntos de la función requerida del programa deseado.

c) Un programa existente que ejecute parte o toda la función deseada, pero que tenga otras características que deban ser mejoradas en el nuevo trabajo de desarrollo.

La secuencia de etapas del paradigma de construcción de prototipos se muestra en la Fig. 2.4. Como en todos los métodos de desarrollo de software, la construcción de prototipos comienza con la recolección de los requisitos. Luego se produce un "diseño rápido". El diseño rápido se enfoca sobre la representación de los aspectos del software visibles al usuario (por ejemplo, métodos de entrada y formatos de salida). El diseño rápido conduce a la construcción de un prototipo. El prototipo es evaluado por el cliente/usuario y se utiliza para refinar los requisitos del software a desarrollar. Se produce un proceso interactivo en el que el prototipo es "afinado" para que satisfaga las necesidades del cliente, al mismo tiempo que facilita al que lo desarrolla una mejor comprensión de lo que hay que hacer.

Idealmente, el prototipo sirve como mecanismo para identificar los requisitos de software. Si se va a construir un prototipo que funcione, el desarrollador intenta hacer uso de fragmentos de programas existentes o aplica herramientas (por ejemplo, generadores de reportes, generadores de ventanas, etc.) que faciliten la rápida

generación de programas que funcionen.

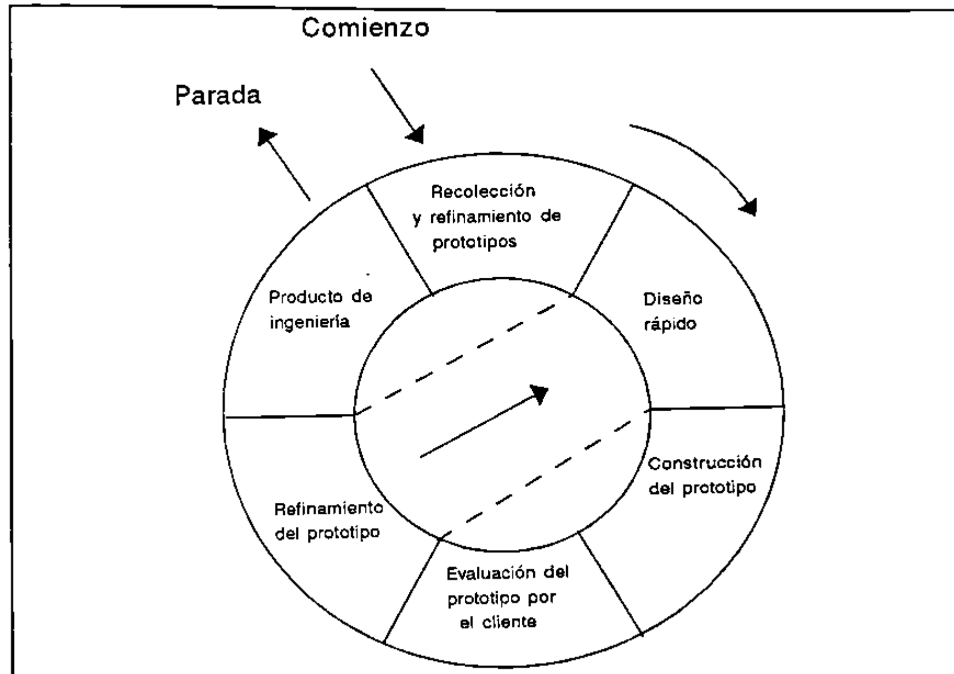


Fig. 2.4 Creación de prototipos.
Fuente: (Pressman, 1993).

El modelo en espiral.

El modelo en espiral para la ingeniería de software ha sido desarrollado para cubrir las mejores características tanto del ciclo de vida clásico, como de la creación de prototipos, añadiendo al mismo tiempo un nuevo elemento: el análisis de riesgo. El modelo representado en la Fig. 2.5, define cuatro actividades principales:

a) Planificación: determinación de objetivos, alternativas y restricciones.

b) Análisis de riesgo: análisis de alternativas e identificación/resolución de riesgos.

c) Ingeniería: desarrollo del producto a un nivel superior.

d) Evaluación del cliente: valoración de los resultados de la ingeniería.

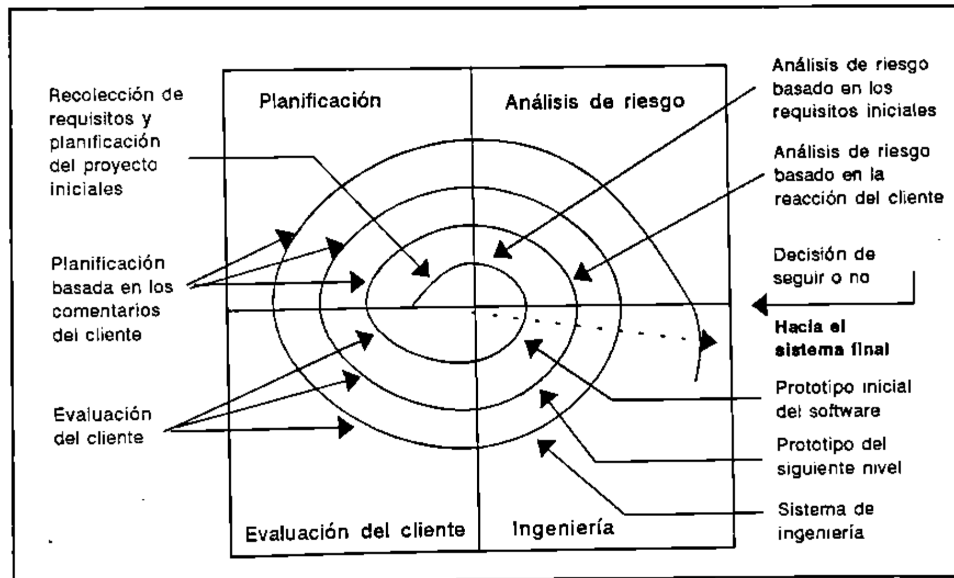


Fig. 2.5 El modelo en espiral.
Fuente: (Pressman, 1993).

En cada iteración alrededor de la espiral (comenzando en el centro y siguiendo hacia el exterior), se construyen versiones sucesivas del software, cada vez más completas.

Durante la primera vuelta alrededor de la espiral se definen los objetivos, las alternativas y las restricciones, y se analizan e identifican los riesgos. Si el análisis de riesgo indica que hay una incertidumbre en los requisitos, se puede usar la creación de prototipos en el cuadrante de ingeniería para dar asistencia tanto al encargado del desarrollo como al cliente.

El cliente evalúa el trabajo de ingeniería (cuadrante de evaluación del cliente) y sugiere modificaciones. En base a los comentarios del cliente se produce la siguiente fase de planificación y de análisis de riesgo. En cada bucle alrededor de la espiral, la culminación del análisis de riesgo resulta en una decisión de "seguir o no seguir". Si los riesgos son demasiados grandes, se puede dar por terminado el proyecto.

Si no se da por terminado el proyecto, se sigue avanzando alrededor del camino de la espiral, y ese camino lleva a los desarrolladores hacia afuera, hacia un modelo más completo del sistema, y, al final, al propio sistema operacional. Cada vuelta alrededor de la espiral requiere ingeniería (Fig. 2.5 cuadrante inferior derecho), que se puede llevar a cabo mediante el enfoque del ciclo de vida clásico o de la creación de prototipos.

2.2.3 UNIFICACION DE LOS PARADIGMAS.

Independientemente del paradigma de ingeniería elegido, el proceso de desarrollo de software contiene tres fases genéricas: definición, desarrollo y mantenimiento.

La fase de definición.

La fase de definición se centra sobre el qué. Esto es, durante la definición, el que desarrolla el software intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué interfases han de establecerse, qué restricciones de diseño existen y qué criterios de validación se necesitan para definir un sistema correcto. Por lo tanto, han de identificarse los requisitos clave del sistema y del software.

Los pasos específicos en la fase de definición son los siguientes:

a) Análisis del sistema. El análisis del sistema define el papel de cada elemento de un sistema de información, asignando finalmente al software el papel que va a desempeñar.

b) Planificación del proyecto de software. Una vez establecido el ámbito del software, se analizan los riesgos, se asignan los recursos, se estiman los costos, se definen

las tareas y se planifica el trabajo.
c) Análisis de requisitos. El ámbito establecido para el software proporciona la dirección a seguir, pero antes de comenzar a trabajar es necesario disponer de una información más detallada del ámbito de información y de función del software.

La fase de desarrollo.

La fase de desarrollo se centra en el cómo. Esto es, durante esta fase, el que desarrolla el software intenta descubrir cómo han de diseñarse las estructuras de datos y la arquitectura del software, cómo han de implementarse los detalles procedimentales, cómo ha de traducirse el diseño a un lenguaje de programación (o lenguaje no procedimental) y cómo ha de realizarse la prueba.

Los pasos específicos en la fase de desarrollo son los siguientes:
a) Diseño del software. El diseño traduce los requisitos del software a un conjunto de representaciones (algunas gráficas y otras tabulares o basadas en lenguajes) que describen la estructura de los datos, la arquitectura, el procedimiento algorítmico y las características de la interfase.
b) Codificación. Las representaciones del diseño deben ser traducidas a un lenguaje artificial, dando como resultado

instrucciones ejecutables por la computadora. El paso de la codificación es el que lleva a cabo esa traducción.

c) Prueba del software. Una vez que el software ha sido implementado en una forma ejecutable por la máquina, debe ser probado para descubrir los defectos que pueden existir en la función, en la lógica y en la implementación.

La fase de mantenimiento.

La fase de mantenimiento se centra en el cambio que va asociado a la corrección de errores, a las adaptaciones requeridas por la evolución del entorno del software y a las modificaciones debidas a los cambios de los requisitos del cliente dirigidos a reforzar o ampliar el sistema.

Durante la fase de mantenimiento se encuentran tres tipos de cambios:

a) Corrección. Es muy probable que el cliente descubra defectos en el software, aunque se hayan llevado a cabo las mejores actividades de garantía de calidad. El mantenimiento correctivo cambia el software para corregir los defectos.

b) Adaptación. Con el paso del tiempo es probable que cambie el entorno original (por ejemplo, los requisitos de la aplicación, el CPU, el sistema operativo, los periféricos) para el cual se desarrolló el software. El mantenimiento adaptativo consiste en modificar el software para acomodarlo

a los cambios de su entorno externo.

c) Mejora. Conforme utilice el software, el cliente/usuario puede descubrir funciones adicionales que pudieran interesarle que estuvieran incorporadas en el software. El mantenimiento perfectivo amplía el software más allá de sus requisitos funcionales originales.

2.3 METRICAS DE SOFTWARE

La medición es muy común en el mundo de la ingeniería. Medimos potencias de consumo, pesos, dimensiones físicas, temperaturas, voltajes, etc. Desafortunadamente, la medición se aleja de lo común en el mundo de la ingeniería de software. Encontramos dificultades en ponernos de acuerdo sobre qué medir y cómo evaluar las medidas.

Algunas de las razones para medir el software son las siguientes:

1) Para indicar la calidad del producto.

2) Para evaluar la productividad de la gente que desarrolla el producto.

3) Para evaluar los beneficios (en términos de productividad y calidad) que se derivan del uso de nuevos métodos y herramientas de la ingeniería de software.

4) Para establecer una línea de base para la estimación.

5) Para ayudar a justificar el uso de nuevas herramientas o

la formación adicional de otras herramientas.

Las mediciones de software pueden clasificarse en dos categorías: medidas directas y medidas indirectas. Entre las medidas directas se encuentran: el costo y el esfuerzo requerido para construir el software, el número de líneas de código, etc. Entre las medidas indirectas del proceso de ingeniería de software se encuentran: la calidad y funcionalidad del software, la eficiencia y facilidad de mantenimiento.

El campo de las métricas del software se pueden clasificar en dos dimensiones (Fig. 2.6). En una dimensión tenemos las siguientes métricas:

1) Métricas de productividad. Estas métricas se centran en el rendimiento del proceso de la ingeniería de software.
2) Métricas de calidad. Estas métricas proporcionan una indicación de cómo se ajusta el software a los requisitos implícitos y explícitos del cliente.
3) Métricas técnicas. Estas métricas se centran en las características del software más que en el proceso a través del cual ha sido desarrollado el software.

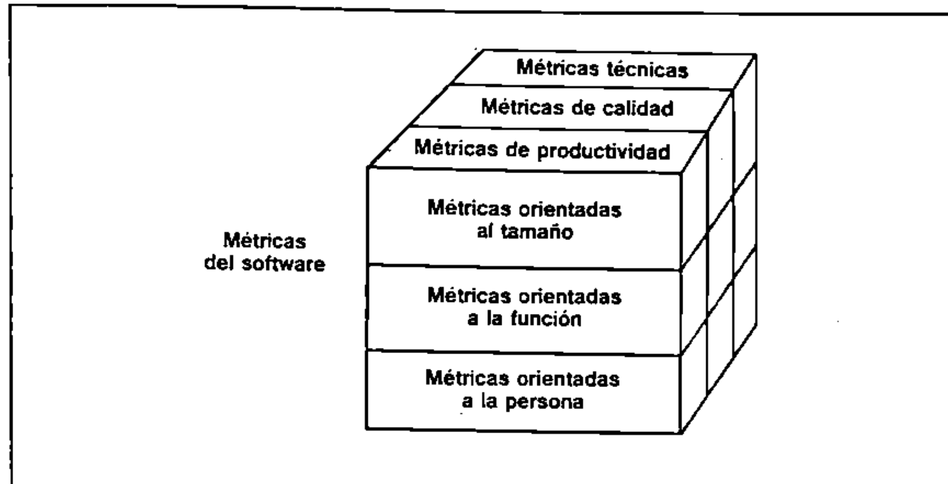


Fig. 2.6 Clasificación de métricas.
Fuente: (Pressman, 1993)

En otra dimensión tenemos las siguientes métricas:

1) Métricas orientadas al tamaño. Estas métricas son medidas directas del software y del proceso por el cual se desarrolla. Por ejemplo; líneas de código (LOC), esfuerzo y costo.

2) Métricas orientadas a la función. Estas métricas son medidas indirectas del software y del proceso por el cual se desarrolla. En lugar de calcular las LOC, las métricas orientadas a la función se centran en la funcionalidad o utilidad del programa.

3) Métricas orientadas a la persona. Estas métricas proporcionan información sobre la forma en que la gente desarrolla el software y sobre el punto de vista humano de la efectividad de las herramientas y los métodos.

2.4 METRICAS DE CALIDAD DEL SOFTWARE

2.4.1 CALIDAD DEL SOFTWARE.

Pressman (Pressman, 1983, p. 576) define la calidad del software como:

"Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente."

La definición de calidad del software hace énfasis en tres puntos importantes:
1) Los requisitos (funcionales) del software son la base de las medidas de la calidad. La falta de concordancia con los requisitos es una falta de calidad.
2) Los estándares especificados definen un conjunto de criterios de desarrollo que guían la forma en que se aplica la ingeniería de software. Si no se siguen esos criterios, casi siempre habrá falta de calidad.
3) Existe un conjunto de requisitos implícitos que a menudo no se mencionan (por ejemplo, el deseo de un buen mantenimiento). Si el software se ajusta a sus requisitos explícitos pero falla en alcanzar los requisitos implícitos,

la calidad del software daja mucho que desear.

2.4.2 FACTORES DE CALIDAD DEL SOFTWARE.

Los factores que afectan la calidad del software se clasifican en dos categorías:

1) Factores que pueden ser medidos directamente (por ejemplo, errores/LOC/unidad de tiempo).
2) Factores que sólo pueden ser medidos indirectamente (por ejemplo, facilidad de uso o de mantenimiento).

Una clasificación de los factores que afectan la calidad del software ha sido propuesta por McCall (1977). Estos factores de calidad del software, que se pueden ver en la Fig. 2.7, se centran en tres aspectos importantes de un producto de software: sus caracteristicas operativas, su capacidad de soportar los cambios y su adaptabilidad de nuevos entornos.

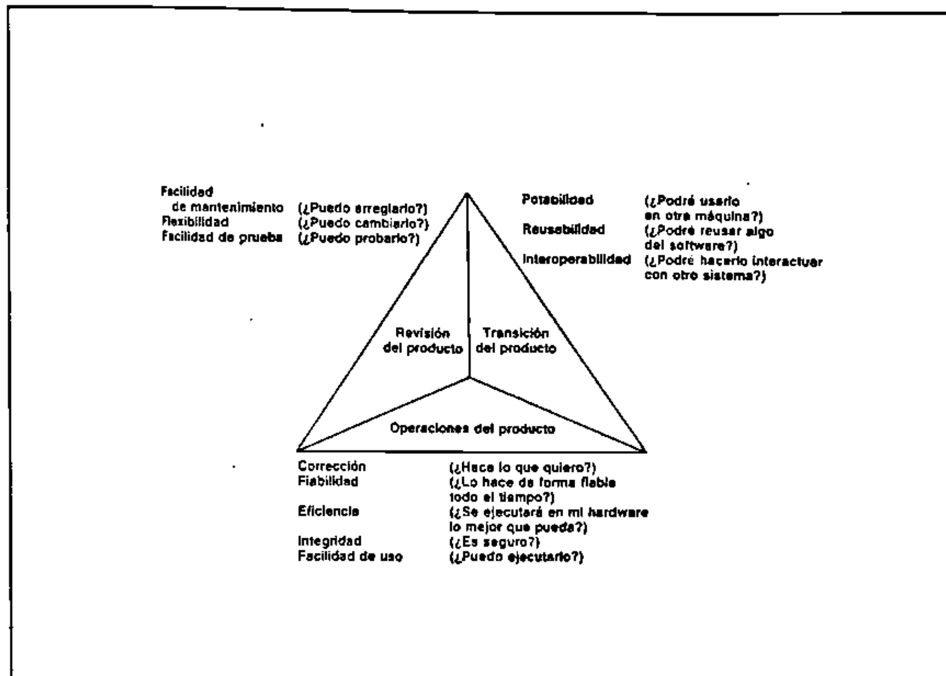


Fig. 2.7 Factores de calidad del software de McCall.
Fuente: (Pressman, 1993).

Es difícil, y en algunos casos imposible, desarrollar medidas directas de los anteriores factores de calidad. Por tanto, se define un conjunto de métricas usadas para medir en forma indirecta los factores de calidad del software. Existen dos tipos de métricas:

1) Métricas cualitativas. Estas métricas sólo pueden ser medidas en forma subjetiva.

2) Métricas cuantitativas. Estas métricas sí pueden medirse en forma objetiva.

2.4.3 METRICAS CUALITATIVAS DE LA CALIDAD DEL SOFTWARE.

Entre las métricas cualitativas de la calidad del software encontramos las siguientes métricas definidas por McCall (McCall, 1977):

Facilidad de auditoría. La facilidad con que se puede comprobar la conformidad con los estándares.

Exactitud. La precisión de los cálculos y del control.

Normalización de las comunicaciones. El grado en que se usan el ancho de banda, los protocolos, y las interfases estándar.

Completitud. El grado en que se ha conseguido la total implementación de las funciones requeridas.

Concisión. Lo compacto que es el programa en términos de líneas de código.

Consistencia. El uso de un diseño uniforme y de técnicas de documentación a lo largo del proyecto de desarrollo de software.

Estandarización en los datos. El uso de estructuras de datos y de tipos estándar a lo largo de todo el programa.

Tolerancia de errores. El daño que se produce cuando el programa encuentra un error.

Eficiencia en la ejecución. El rendimiento en tiempo de ejecución de un programa.

Facilidad de expansión. El grado en que se puede ampliar el diseño arquitectónico, de datos o procedimental.

Generalidad. La amplitud de aplicación potencial de los componentes del programa.

Independencia del hardware. El grado en que el software es independiente del hardware sobre el que opera.

Instrumentación. El grado en que el programa muestra su propio funcionamiento e identifica errores que aparecen.

Modularidad. La independencia funcional de las componentes del programa.

Facilidad de operación. La facilidad de operación de un programa.

Seguridad. La disponibilidad de mecanismos que controlen o protejan los programas o los datos.

Autodocumentación. El grado en que el código fuente proporciona documentación significativa.

Simplicidad. El grado en que un programa puede ser entendido sin dificultad.

Independencia del sistema de software. El grado en que el programa es independiente de características no estándar del lenguaje de programación, de las características del sistema operativo y de otras restricciones del entorno.

Facilidad de traza. La posibilidad de rastrear la representación del diseño o de los componentes reales del programa hacia atrás, hacia los requisitos.

Formación. El grado en que el software ayuda para permitir que nuevos usuarios apliquen el sistema.

El esquema de graduación para estas métricas, propuesto por McCall (1977), va en una escala de 0 (bajo) a 10 (alto). La relación entre los factores de calidad del software y las métricas cualitativas de calidad se muestran en la Tabla 2.1. El peso dado en cada métrica dependerá de los productos particulares, y de otros aspectos.

Factor de calidad \ Métrica de calidad del software	Corrección	Fiabilidad	Eficiencia	Integridad	Facilidad de mantenimiento	Flexibilidad	Facilidad de prueba	Portabilidad	Reusabilidad	Facilidad de interoperación	Facilidad de uso
Facilidad de auditoria				x			x				
Exactitud		x									
Norm. de las comunicaciones										x	
Completitud	x										
Complejidad		x				x	x				
Concisión			x		x	x					
Consistencia	x	x			x	x					
Estandarización en los datos										x	
Tolerancia de errores		x									
Eficiencia en la ejecución			x								
Facilidad de expansión						x					
Generalidad						x		x	x	x	
Independencia del hardware								x	x		
Instrumentación				x	x		x				
Modularidad		x			x	x	x	x	x	x	
Facilidad de operación			x								x
Seguridad				x							
Autodocumentación					x	x	x	x	x		
Simplicidad		x			x	x	x				
Indep. del sistema de software								x	x		
Facilidad de traza	x										

Tabla 2.1 Factores y métricas de calidad.
Fuente: (Pressman, 1993).

2.4.4 METRICAS CUANTITATIVAS DE LA CALIDAD DEL SOFTWARE.

Algunas de las métricas cuantitativas de la calidad del software son:

1) Indices de calidad del software.

2) Medida de Complejidad de McCabe.

3) Métricas de Halstead.

Indices de calidad del software.

El US Air Force Systems Command ha desarrollado una serie de indicadores de calidad del software basadas en las características de diseño medibles para un programa de computadora. La Air Force utiliza información obtenida a partir del diseño arquitectónico y de datos, para obtener un índice de calidad de la estructura del diseño, cuyo valor está entre 0 y 1.

El estándar de IEEE 982.1 sugiere un índice de madurez del software, el cual proporciona una indicación de la estabilidad de un producto de software (basada en los cambios que se producen en cada versión del producto).

Medida de complejidad de McCabe.

Esta medida propuesta por Thomas McCabe (McCabe, 1976) se basa en la representación del flujo de control de un programa. Para describir el flujo de control se usa una red del programa (Fig. 2.8). Cada círculo representa una tarea de procesamiento (una o más sentencias de código); el flujo

de control (ramificaciones) se representa mediante flechas correctivas.

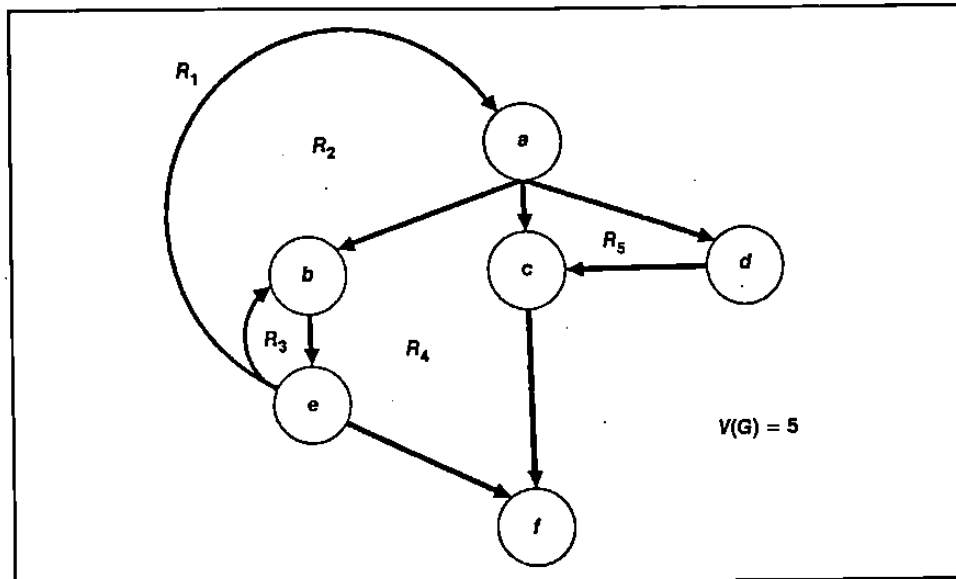


Fig. 2.8 Complejidad de la red de flujo de control.
Fuente: (Pressman, 1993).

McCabe (1976) define una medida de la complejidad del software que se basa en la complejidad ciclomática de la red del programa de un módulo. Una técnica que se puede usar para calcular la métrica de complejidad ciclomática, es la de determinar el número de regiones que forma la red en el plano.

Métricas de Halstead.

La teoría de Halstead sobre la ciencia del software asigna leyes cuantitativas al desarrollo de software de

computadora. La teoría de Halstead se deriva de la suposición fundamental: "El cerebro humano sigue un conjunto de reglas más rígido (al desarrollar algoritmos) del que nunca ha tenido consciencia..." (Halstead, 1977, p. 15). En esta teoría se usa un conjunto de medidas básicas que se pueden obtener una vez que se ha generado el código fuente o estimar una vez que se ha terminado el diseño.

2.5 RESUMEN

En este capítulo se ha presentado una breve explicación del lugar en que se ubican las métricas de Halstead, así como de los conceptos que nos conducen hacia ellas. En el siguiente capítulo se analizarán en forma más detallada las métricas de Halstead.

CAPITULO 3

METRICAS DE HALSTEAD Y PREGUNTAS DE LA INVESTIGACION

3.0 INTRODUCCION

En el capítulo anterior ubicamos las métricas de Halstead dentro de la disciplina de la Ingeniería de Software. Este capítulo se divide en dos partes fundamentales. La primera parte trata la descripción explícita de las métricas de Halstead, en base a la Teoría de la Ciencia del Software (Halstead, 1977), como modelo teórico de la investigación. La segunda parte trata las preguntas e hipótesis de la investigación.

El presente capítulo se dirige de la siguiente forma. En la sección 3.1 se comenta la importancia que tienen las métricas de Halstead en distintas áreas. En 3.2 se definen los conceptos básicos de la ciencia del software. En 3.3 se explica el estimador de longitud de un programa. En 3.4 se explica el estimador del volumen potencial de un programa. En 3.5 se explica el estimador del nivel/dificultad y se define el contenido de inteligencia de un programa. En 3.6 se explican el estimador del esfuerzo y del tiempo de programación de un programa. En 3.7 se define el nivel del lenguaje de un programa y se explica su estimador. En 3.8 se

proponen las preguntas e hipótesis de la investigación. En 3.9 se hace un resúmen del capítulo.

3.1 IMPORTANCIA

Aunque ha habido muchas críticas a las métricas de Halstead (Shen, 1983), no cabe duda que su aplicación ha tenido efecto en muchas áreas. En el área de educación se ha demostrado que los estudiantes que han recibido una enseñanza en estas métricas (y otras) producen programas que exhiben menos complejidad, requieren menos tiempo de codificación y prueba, y además, es más fácil de darle mantenimiento (Bowman, 1990). En el área de sistemas de información, las métricas de Halstead se han utilizado para estimar el tamaño de un sistema de información (Wrigley, 1991).

3.2 DEFINICIONES BASICAS

Un programa computacional se considera, en la ciencia del software, como una serie de partículas elementales que pueden ser clasificadas como operadores ú operandos. Esto se basa en el hecho de que todos los programas pueden ser reducidos en una sucesión de instrucciones de lenguaje máquina, cada una de las cuales contiene un operador y un número de direcciones de operandos. Todas las medidas de la

ciencia del software están en función de la cantidad de estas partículas elementales. Las métricas básicas se definen como:

n_1 = número de operadores diferentes. (1)

n_2 = número de operandos diferentes. (2)

N_1 = número total de operadores. (3)

N_2 = número total de operandos. (4)

Generalmente, se considera operador cualquier símbolo en un programa que especifíque una acción algorítmica, mientras que un símbolo usado para representar datos se considera un operando. La mayoría de las señales de puntuación se clasifican como operadores. El vocabulario de un programa, que consiste del número de las diferentes partículas elementales usadas para construir un programa, se define como:

$$n = n_1 + n_2 \qquad (5)$$

La longitud de un programa, en términos del número total de partículas elementales usados, se define como:

$$N = N_1 + N_2 \qquad (6)$$

Obsérvese que N está estrechamente relacionada con la tradicional medida de líneas de código (LOC) de la longitud

de un programa. Para los programas en lenguaje máquina donde cada una de las líneas consiste de un operador y de un operando, se tiene que $N = 2 \times LOC$.

Se definen adicionalmente otras métricas usando estos términos básicos. Una métrica interesante para el tamaño del programa es la llamada volumen:

$$V = N \times \log_2(n) \qquad (7)$$

La unidad de medida del volumen es la unidad común para tamaño, a saber, "bits". El volumen es el tamaño actual de un programa en una computadora si se utiliza una codificación binaria uniforme para el vocabulario.

El volumen también se puede interpretar como el número de comparaciones mentales que se necesitan para escribir un programa de longitud N, suponiendo que se usa un método de inserción binaria para seleccionar un miembro del vocabulario de tamaño n.

Como el algoritmo se puede implementar de diferentes maneras, pero en programas equivalentes, un programa que tiene el tamaño mínimo se dice que tiene el volumen potencial, el cual denotaremos por V^*.

Cualquier programa con volumen V se considera que se implementa en el nivel del programa L, el cual se define por:

$$L = V^{*}/V \qquad (8)$$

El valor de L se encuentra en el intervalo semiabierto (0,1], donde L = 1 representa un programa escrito en el más alto nivel posible (i.e., con tamaño mínimo).

Lo inverso del nivel del programa se llama dificultad. Esto es:

$$D = 1/L \qquad (9)$$

Cuando el volumen de una implementación de un programa crece, el nivel del programa decrece y la dificultad se incrementa. De este modo, las prácticas de programación tales como el uso redundante de operandos, o el error de usar frases de control de nivel más alto tenderán a incrementar el volumen así como la dificultad.

El esfuerzo que se requiere para implementar un programa de computadora se incrementa cuando el tamaño del programa crece. También toma más esfuerzo implementar un programa en un nivel más bajo (dificultad más alta) comparado con otro programa equivalente en un nivel más alto

(dificultad más baja). Por ejemplo, toma más esfuerzo implementar un programa que despliegue una pantalla en PASCAL que implementarlo en FOXPRO2. De ese modo, se define el esfuerzo en la ciencia del software como:

$$E = V/L = D \times V \qquad (10)$$

La unidad de medida de E es "discriminaciones binarias mentales elementales" (Halstead, 1977).

3.3 ESTIMADOR DE LA LONGITUD DE UN PROGRAMA

La primera hipótesis de la ciencia del software es que la longitud de un programa bien estructurado solamente es una función del número de operadores y operandos diferentes. Esta es llamada la ecuación de longitud, donde Nest es la longitud estimada del programa.

$$Nest = n_1 \log_2(n_1) + n_2 \log_2(n_2) \qquad (11)$$

Hay evidencia empírica de que Nest es un estimador aceptable de N cuando se aplica a un amplio rango de programas escritos en Fortran, Cobol, y PL/S (Shen, 1983). Estos lenguajes son de los llamados de tercera generación.

3.4 ESTIMADOR DEL VOLUMEN POTENCIAL DE UN PROGRAMA

Como se discutió antes, un programa que implementa un algoritmo en su forma más breve tiene el volumen potencial V^*. Si la función que se desea está ya definida en el lenguaje de programación o en su librería de subrutinas como un procedimiento, el volumen potencial se logra especificando el nombre del procedimiento y dando una lista de parámetros entrada/salida. El vocabulario de este programa consiste de dos operadores y de n_2^* operandos. Uno de los operadores es el nombre del procedimiento, ya que define una acción; el otro operador es un símbolo de agrupamiento que se necesita para separar la lista de parámetros del nombre de procedimientos. De este modo,

$$V^* = (2 + n_2^*)\log_2(2 + n_2^*) \qquad (12)$$

donde n_2^* es el número de parámetros entrada/salida para este procedimiento. Esta fórmula, aunque útil para muchos programas, no es aplicable universalmente, ya que hay programas que no tienen una lista explícita de parámetros entrada/salida. Por ejemplo, muchos de los programas escritos en FOXPRO2 o DBASE III, no traen consigo esta lista de parámetros.

3.5 ESTIMADOR DEL NIVEL/DIFICULTAD DE UN PROGRAMA Y DEFINICION DEL CONTENIDO DE INTELIGENCIA DE UN PROGRAMA

El nivel (8) de una implementación particular depende de la razón entre el volumen potencial y el volumen actual. Ya que el volumen potencial, usualmente, no está disponible, una fórmula alternativa que estima el nivel se define como:

$$Lest = 1/Dest = (2/n_1)(n_2 / N_2) \qquad (13)$$

Un argumento intuitivo para esta fórmula es que la dificultad de la programación se incrementa si se introducen operadores extra (aumenta $n_1/2$) y si un operando se usa repetidamente (aumenta N_2/n_2). Cada parámetro en (13) se puede obtener contando los operadores y operandos en un programa computacional. El volumen potencial V^*, entonces, se puede deducir usando (8) con L igual a Lest.

La ecuación (8) sugiere que para un algoritmo dado, diferentes implementaciones pudieran tener volumenes y niveles diferentes, mientras que el producto de estos dos pudiera mantenerse constante. Esto es, el volumen potencial $V^* = L x V$ depende solamente del algoritmo, no sobre las características de una implementación particular.

Cuando Lest de (13) se utiliza para estimar L, el producto (Lest)x(V) es llamado contenido de inteligencia I, esto es,

$$I = (Lest)xV \qquad (14)$$

Se espera, también, que el contenido de inteligencia I se mantenga constante en diferentes implementaciones del mismo problema, ya que es un estimador de V^*.

3.6 ESTIMADOR DEL ESFUERZO Y DEL TIEMPO DE PROGRAMACION DE UN PROGRAMA

Una de las mayores demandas por la ciencia del software es su capacidad para relacionar sus métricas básicas al tiempo común de implementación. El psicólogo, J. Stroud, sugirió que el humano es capaz de hacer un número limitado de discriminaciones elementales por segundo (Halstead, 1977). Stroud afirmó que este número S (ahora llamado número de Stroud) está clasificado entre 5 y 20. Como el esfuerzo E tiene como unidad de medida el "número de discriminaciones mentales elementales", el tiempo de programación T, en segundos, de un programa es:

$$T = E/S \qquad (15)$$

S está normalmente colocado en 18, ya que ésto pareció dar el mejor resultado en los experimentos de Halstead comparando los tiempos de predicción con los tiempos de programación observados -que incluyeron: tiempo de diseño, codificación y prueba.

De (10) tenemos que E = V/L, pero utilizando la estimación de L, Lest en (13), y V de (7) tenemos que el estimador del esfuerzo Eest es

$$Eest = (n_1 x N_2 x N x \log_2(n))/(2n_2) \qquad (16)$$

y por lo tanto, de (15) y (16), el tiempo estimado de programación es:

$$Test = (n_1 x N_2 x N x \log_2(n))/(36n_2) \qquad (17)$$

La ciencia del software sostiene que esta fórmula puede ser usada para estimar el tiempo de programación cuando un problema dado es resuelto por un solo programador hábil y concentrado que escribe un programa de un solo módulo.

3.7 DEFINICION DEL NIVEL DEL LENGUAJE DE PROGRAMACION Y SU ESTIMADOR

La proliferación de lenguajes de programación sugiere la necesidad de una métrica que exprese el poder de un lenguaje. Halstead hipotetizó que si el lenguaje de programación se mantiene fijo, entonces mientras V^* crece, L decrece de tal forma que el producto L x V^* se mantiene constante. Así, este producto llamado nivel del lenguaje (λ), se puede usar para caracterizar un lenguaje de programación. Esto es,

$$\lambda = L \times V^* = L^2 V \qquad (18)$$

sustituyendo, Lest por L de la ecuación (13) y V de la ecuación (7), tenemos que el estimador para el nivel del lenguaje λest es:

$$\lambda est = [(2/n_1)(n_2 /N_2)]^2 x N x \log_2(n) \qquad (19)$$

Analizando un número de programas diferentes escrito en lenguajes diferentes, se determinó los niveles de lenguaje para cada uno de estos lenguajes (Tabla 3.1).

Lenguaje	λ	σ^2
Inglés	2.16	0.74
PL/I	1.53	0.92
Algol 58	1.21	0.74
Fortran	1.14	0.81
Pilot	0.92	0.43
Assembly	0.88	0.42

Tabla 3.1 Media y varianza del nivel del lenguaje para seis lenguajes.
Fuente: (Halstead, 1977).

Estos valores promedio obedecen a la mayoría de las clasificaciones intuitivas de los programadores para estos lenguajes, pero todos ellos tienen grandes varianzas. Tales fluctuaciones en un valor fijo hipotetizado no son completamente inesperadas ya que el nivel del lenguaje no solo depende del lenguaje en sí mismo, sino también de la naturaleza del problema que está siendo programado así como de la habilidad y estilo del programador.

En la Tabla 3.2 se puede ver las distribuciones de frecuencia de los niveles de lenguaje para diferentes lenguajes.

λ	Comp	Pilot	Fort	Alg 58	PL/I	Inglés
(0.00,0.49)	14%	14%	14%	14%	14%	0%
(0.50,0.99)	57	64	43	29	29	0
(1.00,1.49)	14	14	29	36	21	23
(1.50,1.99)	14	0	7	14	14	17
(2.00,2.49)	0	7	0	0	21	27
(2.50,2.99)	0	0	0	0	0	17
(3.00,3.49)	0	0	0	7	0	10
(3.50,3.99)	0	0	7	0	7	7

Tabla 3.2 Distribuciones de frecuencia de niveles de lenguaje para diferentes lenguajes.
Fuente: (Halstead,1977).

3.8 PROPUESTA DE PREGUNTAS E HIPOTESIS DE INVESTIGACION

Como se comentó en la sección anterior, hay evidencia empírica de que el estimador de longitud de un programa (Nest) es un buen estimador de la longitud del programa (N) para lenguajes de tercera generación, desde un punto de vista estadístico. En consecuencia, podemos proponer nuestra primer pregunta de investigación: ¿Es el estimador Nest un buen estimador de N, en lenguajes de cuarta generación? Con esta pregunta, formulamos nuestra primer hipótesis de investigación:

H1: Para lenguajes de cuarta generación, Nest = N.

Por otra parte, se espera que el nivel del lenguaje de los 4GL's sea mayor que el nivel del lenguaje de los 3GL's,

pero tiene que ser menor que el nivel del lenguaje natural (Inglés prosaico). Esto nos conduce a la siguiente pregunta de investigación, ¿Es el nivel del lenguaje de los 4GL's mayor que el nivel del lenguaje de los 3GL's y menor que el nivel del lenguaje natural? Con esta pregunta, formulamos la segunda hipótesis de investigación:

H2: Para lenguajes de cuarta generación, el nivel del lenguaje es mayor que 1.53 y menor que 2.16.

3.9 RESUMEN

En este capítulo se presentó el modelo teórico en el que se basa esta investigación, y se propusieron las preguntas e hipótesis de la investigación. En el siguiente capítulo discutiremos el enfoque de la investigación que se tomó para investigar las preguntas propuestas.

CAPITULO 4

METODOLOGIA DE LA INVESTIGACION

4.0 INTRODUCCION

En el capítulo anterior se discutió el modelo teórico de la investigación y se propusieron las preguntas e hipótesis de la investigación. En este capítulo se discute la metodología empleada en la investigación.

El presente capítulo se dirige de la siguiente forma. En la sección 4.1 se discute la selección del diseño de investigación. En la 4.2 se discute la selección de los 4GL's. En la sección 4.3 se discute la selección de la muestra. En 4.4 se discute el desarrollo y validación del instrumento de medición. En 4.5 se hace un resúmen del capítulo.

4.1 DISEÑO DE INVESTIGACION

La investigación se llevó a cabo como un estudio ex post facto usando un analizador de código para programas en código fuente como instrumento para recolectar los datos. Se seleccionó un estudio ex post facto debido a la naturaleza de la investigación.

El diseño ex post facto (Emory, 1991) es una clase de diseño de investigación, en el cual los investigadores no tienen control sobre las variables en el sentido de ser capaces de manipularlas. Los investigadores sólamente reportan lo que ha pasado o lo que está pasando. De hecho, es importante en este diseño que los investigadores no tengan influencia en las variables, ya que al hacer ésto introduce sesgo en los resultados.

4.2 SELECCION DE LOS 4GL's

Los 4GL's que se seleccionaron para hacer el estudio fueron los lenguajes DBASE III y FOXPRO2. La selección de estos lenguajes se debe a las siguientes razones:
1) En el ambiente regional, son dos de los 4GL's más utilizados actualmente en la mediana empresa (Crespo, 1992).
2) La sintáxis de ambos 4GL's es muy parecida.

4.3 SELECCION DE LA MUESTRA

Una de las consideraciones más importantes que nos conducen a obtener resultados más confiables al evaluar las métricas de Halstead, es eliminar la introducción de impurezas en los programas (Halstead, 1977). Esto es, el programa debe estar escrito con una buena programación.

Para que la consideración anterior se cumpla de la mejor manera posible, se debe seleccionar la muestra de tal forma que garantice que los programas fueron escritos por programadores expertos. Por lo tanto, se seleccionaron como muestras los programas en código fuente que vienen como ejemplos en los paquetes de FOXPRO2 y DBASE III. En total fueron seleccionados 117 programas para FOXPRO2 y 26 para DBASE III.

4.4 DESARROLLO Y VALIDACION DEL ANALIZADOR DE CODIGO

El instrumento de coleccionar los datos empleados en el estudio fué un analizador de código fuente. Este analizador de código es la parte medular del desarrollo de la investigación, y está construído de tal forma que es capaz de hacer análisis de código para cualquier lenguaje haciendo muy pocas modificaciones. Todo lo relacionado con la parte operativa del analizador de código puede verse en el apéndice A. Para nuestra investigación, el analizador de código se alimentó con programas fuente escritos en FOXPRO2 o en DBASE III. La salida del analizador son las métricas de Halstead.

La validación del analizador no se hizo de una manera formal. Para validar el analizador se escogió una muestra de 20 programas pequeños (hasta 20 líneas de código fuente),

33035

los cuales fueron alimentados al analizador. El resultado se comparó con el resultado de un proceso manual. No se encontró alguna diferencia entre ambos resultados. Así, el analizador está midiendo exactamente lo que queremos medir.

4.5 RESUMEN

En este capítulo discutimos la selección del diseño de investigación, identificamos los 4GL's que vamos a analizar, así como la selección de la muestra y el instrumento de medición. En el siguiente capítulo analizaremos los datos para contestar las preguntas e hipótesis de la investigación.

CAPITULO 5

ANALISIS DE DATOS

5.0 INTRODUCCION

Este capítulo presenta el análisis de datos que se efectuó para responder a las hipótesis de investigación. El capítulo se dirige de la siguiente forma. En la sección 5.1 se revisa la exactitud de los datos. En 5.2 se presenta un resúmen de estadísticas descriptivas. En 5.3 se analizan los datos para responder la primer pregunta e hipótesis de la investigación. En 5.4 se analizan los datos para responder la segunda pregunta e hipótesis de investigación. En 5.5 se hace un resúmen del capítulo.

5.1 REVISION DE LA EXACTITUD DE LOS DATOS

Como señalamos en el capítulo anterior, necesitamos programas que hayan sido escritos con una buena programación. Sin embargo, encontramos que algunos de los programas que vienen de ejemplo en los paquetes FOXPRO2, Y DBASE III no vienen escritos en forma consistente. Por ejemplo, el comando PARAMETERS es comúnmente escrito como PARAMETER.

La flexibilidad de los paquetes, permite al programador escribir las primeras cuatro letras (al menos) del comando para obtener la ejecución completa del mismo. Por ejemplo, si el programador escribe MODI, MODIF, o MODIFY significa que ejecute el comando MODIFY. Esto hace más difícil el conteo de operadores dentro de un programa.

Aunque se detectaron estos defectos, no fueron corregidos para tratar de darle más realidad a los datos. Por lo tanto, la exactitud de los datos obtenidos, dependen de lo bien que esté escrito el programa.

Por otra parte, el número de operadores que podemos formar dentro de estos lenguajes es demasiado grande. Por ejemplo, sólo el comando BROWSE tiene al menos 16 subcomandos más, lo cual significa alrededor de 100 billones de formas distintas de construir el comando BROWSE (Pinter, 1992). Por lo tanto, es prácticamente imposible escribir cada uno de los comandos que podemos formar en FOXPRO2 o en DBASE III.

La exactitud de los datos también depende del número total de operadores que estamos considerando. En efecto, si un operador no se encuentra en la lista de operadores asignada para el conteo, éste será considerado como un operando. Sin embargo, se trató de eliminar este problema

dando una lista de operadores lo suficientemente grande que involucrara a todos los comandos. Además, en el conteo que se lleva a cabo en el analizador se trata también de eliminar este problema.

5.2 ESTADISTICAS DESCRIPTIVAS

La Tabla 5.1 muestra los valores reales (N) y los valores estimados (Nest) de las longitudes de los programas escritos en DBASE III.

Prog	N	Nest	Prog	N	Nest	Prog	N	Nest
01	29	51	11	330	501	21	749	891
02	56	145	12	332	524	22	778	432
03	70	103	13	352	586	23	819	820
04	96	226	14	368	615	24	988	1026
05	152	263	15	394	628	25	1098	899
06	154	296	16	448	640	26	1391	1271
07	159	239	17	450	537			
08	184	343	18	459	551			
09	189	328	19	524	361			
10	247	359	20	634	861			

Tabla 5.1 Longitudes reales y estimadas para DBASE III.

La Tabla 5.2 muestra los valores reales (N) y los valores estimados (Nest) de las longitudes de los programas escritos en FOXPRO2.

Prog	N	Nest	Prog	N	Nest	Prog	N	Nest
01	12	27	40	81	92	79	207	367
02	22	44	41	83	134	80	214	480
03	24	58	42	84	232	81	216	419
04	25	40	43	84	129	82	219	525
05	26	48	44	85	196	83	221	259
06	27	56	45	86	232	84	222	325
07	30	62	46	90	163	85	225	399
08	31	48	47	94	257	86	227	359
09	31	49	48	97	117	87	233	366
10	37	91	49	99	102	88	240	455
11	37	62	50	104	226	89	241	363
12	37	62	51	110	64	90	244	336
13	39	71	52	112	173	91	251	251
14	40	76	53	115	226	92	269	388
15	40	76	54	119	202	93	272	363
16	41	84	55	119	129	94	293	475
17	41	81	56	128	239	95	298	461
18	46	67	57	128	198	96	311	477
19	46	76	58	129	156	97	349	622
20	47	84	59	129	152	98	374	571
21	47	67	60	135	269	99	374	232
22	47	118	61	136	303	100	386	606
23	50	108	62	141	284	101	505	579
24	51	101	63	143	228	102	532	605
25	54	91	64	146	381	103	562	1185
26	54	91	65	169	266	104	774	559
27	55	107	66	173	333	105	852	720
28	62	99	67	175	179	106	889	906
29	65	145	68	175	346	107	1034	1650
30	67	156	69	177	446	108	1056	991
31	67	133	70	180	233	109	1065	687
32	69	72	71	181	341	110	1289	1232
33	73	151	72	183	307	111	1791	2364
34	74	146	73	183	145	112	2130	1780
35	74	150	74	188	276	113	2177	1854
36	74	150	75	194	288	114	2238	2732
37	78	87	76	195	346	115	2285	1121
38	78	114	77	198	306	116	3451	3618
39	80	124	78	206	314	117	4610	4434

Tabla 5.2 Longitudes reales y estimadas para FOXPRO2.

Las Tablas 5.3 y 5.4 muestran los valores de los niveles del lenguaje para los programas muestra escritos en DBASE III y en FOXPRO2, respectivamente.

Prog	λ	Prog	λ	Prog	λ	Prog	λ	Prog	λ
01	2.062	07	1.863	13	0.980	19	0.577	25	1.160
02	8.587	08	3.290	14	0.958	20	1.242	26	0.914
03	3.199	09	2.032	15	1.718	21	1.617		
04	2.982	10	1.334	16	1.214	22	5.217		
05	1.215	11	1.955	17	0.655	23	0.759		
06	2.120	12	1.265	18	0.649	24	1.251		

Tabla 5.3 Niveles del lenguaje para DBASE III.

Prog	λ	Prog	λ	Prog	λ	Prog	λ	Prog	λ
01	4.613	26	2.943	51	14.167	76	2.803	101	0.372
02	5.372	27	2.201	52	2.675	77	3.329	102	2.443
03	2.533	28	0.552	53	1.837	78	0.994	103	0.570
04	3.264	29	3.089	54	1.634	79	2.044	104	3.234
05	3.932	30	1.751	55	1.783	80	0.898	105	1.063
06	0.653	31	2.650	56	1.111	81	1.184	106	1.391
07	2.228	32	0.586	57	0.729	82	0.629	107	0.373
08	4.183	33	3.079	58	1.277	83	1.422	108	0.666
09	8.153	34	1.636	59	0.887	84	2.888	109	1.682
10	4.773	35	1.700	60	1.638	85	0.896	110	0.208
11	2.263	36	1.700	61	0.611	86	0.534	111	0.673
12	2.263	37	0.572	62	0.536	87	0.510	112	2.373
13	3.239	38	0.729	63	0.609	88	1.305	113	0.191
14	3.555	39	0.867	64	2.474	89	2.472	114	0.349
15	3.555	40	0.578	65	4.105	90	2.196	115	0.402
16	0.840	41	1.579	66	1.243	91	1.088	116	0.371
17	2.157	42	1.755	67	0.970	92	1.577	117	0.340
18	1.331	43	0.830	68	0.676	93	2.344		
19	2.273	44	4.641	69	1.262	94	2.919		
20	0.963	45	1.675	70	0.887	95	2.756		
21	1.360	46	0.656	71	1.169	96	3.110		
22	4.370	47	2.049	72	1.662	97	1.052		
23	1.242	48	1.009	73	0.407	98	3.207		
24	3.472	49	1.839	74	0.769	99	0.418		
25	2.943	50	2.475	75	0.846	100	7.381		

Tabla 5.4 Niveles del lenguaje para FOXPRO2.

La Tabla 5.5 muestra la media y la desviación estándar para los niveles de lenguaje de DBASE III y FOXPRO2.

LENGUAJE	MEDIA	DESVIACION ESTANDAR
DBASE III	1.9544	1.7039
FOXPRO2	1.9763	1.8112

Tabla 5.5 Media y desviación estándar para los niveles del lenguaje DBASE III y FOXPRO2.

La Tabla 5.6 muestra la Tabla de Frecuencias para el nivel del lenguaje de DBASE III.

Intervalo	Frecuencia	Porcentaje	Porcentaje Acumulado
(0,1]	07	26.92	26.92
(1,2]	11	42.31	69.23
(2,3]	04	15.38	84.62
(3,4]	02	07.69	92.31
(4,5]	00	00.00	92.31
(5,6]	01	03.85	96.15
(6,7]	00	00.00	96.15
(7,8]	00	00.00	96.15
(8,9]	01	03.85	100.00

Tabla 5.6 Tabla de frecuencias para el nivel del lenguaje DBASE III.

La Tabla 5.7 muestra la Tabla de Frecuencias para el nivel del lenguaje de FOXPRO2.

Intervalo	Frecuencia	Porcentaje	Porcentaje Acumulado
(0,1]	40	34.19	34.19
(1,2]	31	26.50	60.68
(2,3]	24	20.51	81.20
(3,4]	12	10.26	91.45
(4,5]	06	05.13	96.58
(5,6]	01	00.85	97.44
(6,7]	00	00.00	97.44
(7,8]	01	00.85	98.29
(8,9]	01	00.85	99.15
(9,10]	00	00.00	99.15
(10,11]	00	00.00	99.15
(11,12]	00	00.00	99.15
(12,13]	00	00.00	99.15
(13,14]	00	00.00	99.15
(14,15]	01	00.85	100.00

Tabla 5.7 Tabla de frecuencias para el nivel del lenguaje FOXPRO2.

5.3 ANALISIS DE DATOS DE LA PRIMERA HIPOTESIS DE INVESTIGACION

La primer pregunta de investigación es: ¿Es el estimador Nest es un buen estimador de N, en lenguajes de cuarta generación? Para responder esta primer pregunta de investigación calculamos el coeficiente de correlación de Pearson entre N y Nest (de la misma manera en que lo hizo Halstead).

El coeficiente de correlación de Pearson mide la fuerza de la relación lineal que existe entre dos variables, representadas por una muestra de datos bivariados (parejas ordenadas). El coeficiente de correlación de Pearson (r) tiene las siguientes características (Kvanli, 1989):

1) $r \in [-1,1]$.
2) Entre más grande sea $|r|$ (valor absoluto de r), más grande es la relación lineal entre las variables.
3) El valor de r cerca de cero, indica que no existe relación lineal entre las variables.
4) El valor de $r = 1$ o $r = -1$, implica que existe un perfecto patrón lineal entre las variables.
5) Los valores $r = 0$, $r = 1$, o $r = -1$ son muy raros en la práctica.

En la Tabla 5.8 se muestran los resultados de los coeficientes de correlación de Pearson para DBASE III y FOXPRO2.

LENGUAJE	r
DBASE	0.91130
FOXPRO2	0.96215

Tabla 5.8 Coeficientes de correlación de Pearson entre N y Nest.

Los resultados del análisis de correlación en la Tabla 5.8 indican que N y Nest están fuertemente correlacionadas. Por lo tanto, podemos concluir que para los lenguajes de cuarta generación DBASE III y FOXPRO, Nest es un buen estimador de N.

5.4 ANALISIS DE DATOS DE LA SEGUNDA HIPOTESIS DE INVESTIGACION

La segunda pregunta de investigación es: ¿Es el nivel del lenguaje de los 4GL's mayor que el nivel del lenguaje de los 3GL's y menor que el nivel del lenguaje natural? Para responder esta segunda pregunta de investigación hicimos la siguiente consideración: la muestra de los niveles de lenguaje para DBASE III es una muestra grande.

Se considera una muestra grande, aquella cuyo número de elementos es mayor que 30. Una muestra se considera muestra chica si su número de elementos es menor o igual que 30 (Kvanli, 1989).

La consideración anterior se debe a lo siguiente:
1) La muestra está muy próxima a ser muestra grande.
2) La muestra para FOXPRO2 es muestra grande.

La investigación hipotetiza que el nivel del lenguaje de los 4GL's es mayor que 1.53, pero menor que 2.16. Estos valores se tomaron así porque el mayor nivel del lenguaje para un lenguaje de tercera generación es 1.53 (PL/I), y 2.16 es el nivel del lenguaje del Inglés Prosaico (Halstead, 1977).

Para demostrar la hipótesis de investigación hicimos pruebas Z para analizar las medias de los niveles del lenguaje. La prueba Z se utiliza (entre otras cosas) para probar hipótesis sobre la media de una población cuando se tiene una muestra grande y para probar hipótesis sobre medias de muestras independientes (Kvanli, 1989).

La Tabla 5.9 muestra los resultados de la prueba Z, para la siguiente paraja de hipótesis:

H_0 : la media del nivel del lenguaje para los 4GL's es menor o igual a 1.53.

H_a : la media del nivel del lenguaje para los 4GL's es mayor a 1.53.

LENGUAJE	Z	nivel de significancia
DBASE III	1.270	.102
FOXPRO2	2.665	.010

Tabla 5.9 Resultados de las pruebas Z entre los 4GL's y los 3GL's.

Los resultados de la Tabla 5.9 indican que para el 4GL FOXPRO2 hay suficiente evidencia que soporte la hiptesis H_a; esto es, hay suficiente evidencia para concluir que la media del nivel del lenguaje para FOXPRO2 es mayor a 1.53. En el caso de DBASE, el resultado de la hipótesis es inconclusa; esto es, no existe suficiente evidencia para concluir que la media del nivel del lenguaje para DBASE III es mayor a 1.53.

La Tabla 5.10 muestra los resultados de la prueba Z, para la siguiente pareja de hipótesis:

H_0 : la media del nivel del lenguaje para los 4GL's es mayor o igual a 2.16.

H_a : la media del nivel del lenguaje para los 4GL's es menor a 2.16.

LENGUAJE	Z	nivel de significancia
DBASE III	-0.61538	0.26760
FOXPRO2	-0.73875	0.23627

Tabla 5.10 Resultados de la prueba Z, entre el nivel del lenguaje 4GL's y el Inglés Prosaico.

Los resultados de la Tabla 5.10 indican que no hay suficiente evidencia que soporte la hiptesis H_a; esto es, no hay suficiente evidencia para concluir que la media del nivel del lenguaje para los 4GL's (FOXPRO2 y DBASE III) sea menor a 2.16.

Finalmente, la Tabla 5.11 muestra los resultados de la prueba Z (comparamos dos medias de muestras grandes independientes) para la siguiente pareja de hipótesis:

H_0: la media del nivel del lenguaje para DBASE III es igual a la media del nivel del lenguaje para FOXPRO2.

H_a: la media del nivel del lenguaje para DBASE III es diferente a la media del nivel del lenguaje para FOXPRO2.

Z	nivel de significancia
-0.1569077	0.840

Tabla 5.11 Resultados de la prueba Z, comparando las medias de los niveles del lenguaje para FOXPRO2 y DBASE III.

El resultado de la Tabla 5.11 indica que no hay suficiente evidencia que soporte la hiptesis H_a; esto es, no hay suficiente evidencia para concluir que la media del nivel del lenguaje para DBASE III sea diferente a la media del nivel del lenguaje para FOXPRO2.

5.5 RESUMEN

En este capítulo se presentó el análisis de los datos que se recolectaron. El resúmen es el siguiente:

1) Hay una fuerte correlación entre Nest y N, para lenguajes de cuarta generación, lo cual indica que Nest es un buen estimador de N.

2) Hay suficiente evidencia estadística que indica que el nivel del lenguaje FOXPRO2, es mayor que el nivel del lenguaje de los 3GL's (representados por el nivel del lenguaje más alto).

3) No hay suficiente evidencia estadística que indique que haya alguna diferencia entre los niveles del lenguaje de cuarta generación DBASE III y FOXPRO2.

4) No hay suficiente evidencia estadística que indique que el nivel del lenguaje de los lenguajes de cuarta generación DBASE III y FOXPRO2 sea menor que el nivel del lenguaje del Inglés Prosaico.

En el capítulo siguiente discutiremos los resultados y las implicaciones de la investigación.

CAPITULO 6

CONCLUSIONES

En este capítulo se discuten los resultados del análisis de datos que fueron presentados en el capítulo anterior. Además, se dan las conclusiones y algunas sugerencias para investigaciones futuras.

Principales objetivos del estudio.

Los principales objetivos del estudio fueron:
1) Determinar si el estimador de la longitud de un programa propuesto por Halstead (1977) es un buen estimador para lenguajes de cuarta generación.
2) Determinar si el nivel del lenguaje de los 4GL's es mayor que el nivel del lenguaje de los 3GL's, pero menor que el nivel del lenguaje natural.

Discusión, conclusión y sugerencias de investigaciones futuras del objetivo 1.

Encontramos que el estimador de la longitud de un programa propuesto por Halstead, es un buen estimador de la longitud de un programa para los 4GL's: DBASE III y FOXPRO2. Esto es válido en el rango de las longitudes de un programa

del lenguaje específico. Esto es, para FOXPRO2 es válido en el rango de 12 a 4610 y para DBASE III es válido en el rango de 29 a 1391.

Aunque el análisis de correlación lineal nos indique que hay una fuerte correlación lineal entre N y Nest, puede verse en las Tablas 5.1 y 5.2 que Nest empieza sobreestimando la longitud del lenguaje y termina subestimandola. Esto se puede ver mejor en la Tabla 5.2 donde existen más datos.

Dado lo anterior, sería muy importante tener muestras de programas cuyas longitudes tuvieran una mejor dispersión y rangos más grandes para hacer un mejor análisis. Aparentemente la relación entre N y Nest tiene una representación gráfica como en la Fig. 6.1. Una de las sugerencias para un futura investigación sería demostrar que la relación entre N y Nest tiene esta forma.

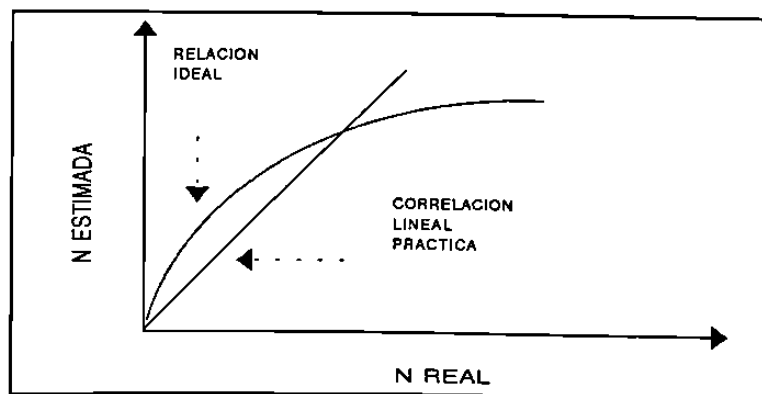


Fig. 6.1 Relación ideal entre N y Nest, y correlación lineal práctica.

Discusión, conclusión y sugerencias de investigaciónes futuras del objetivo 2.

Encontramos que el nivel del lenguaje FOXPRO es mayor que el nivel del lenguaje de los 3GL's. No encontramos soporte estadístico para demostrar que el nivel del lenguaje DBASE III es mayor que el nivel del lenguaje de los 3GL's, pero numéricamente encontramos que sí es mayor con una diferencia de 27.73%. Las posibles causas de no encontar soporte estadístico son el pequeño tamaño de la muestra y la alta desviación estándar.

No encontramos soporte estadístico para demostrar que el nivel del lenguaje DBASE III y FOXPRO2 sea menor que el nivel del lenguaje natural, pero numéricamente encontramos que sí es menor con una diferencia de 9.79% para DBASE III y 8.5% para FOXPRO2. Esto quiere decir que programar en los lenguajes 4GL's (FOXPRO2 y DBASE III) es casi como escribir en Inglés. Más aún, todo parece indicar que no hay diferencia entre los niveles del lenguaje FOXPRO 2 y DBASE III. Esto implica que la selección entre FOXPRO2 y DBASE III no tiene trascendencia con respecto al nivel del lenguaje. Sin embargo, existen otros factores que pueden ayudar a la selección; tales como la experiencia del programador en un lenguaje u otro.

Aunque la clasificación de los niveles del lenguaje FOXPRO2 y DBASE III es la clasificación esperada, parece ser que el nivel del lenguaje para los 4GL's analizados no es constante y por eso las desviaciones estándar son grandes (puede verse en las Tablas 4.3 y 4.4). Aparentemente el nivel del lenguaje depende de la longitud del programa. Más aún, la relación entre el nivel del lenguaje y la longitud del programa idealmente tiene una representación gráfica como se muestra en la Fig. 6.2. Otra de las sugerencias para una futura investigación sería demostrar que la relación entre λ y N tiene esta forma.

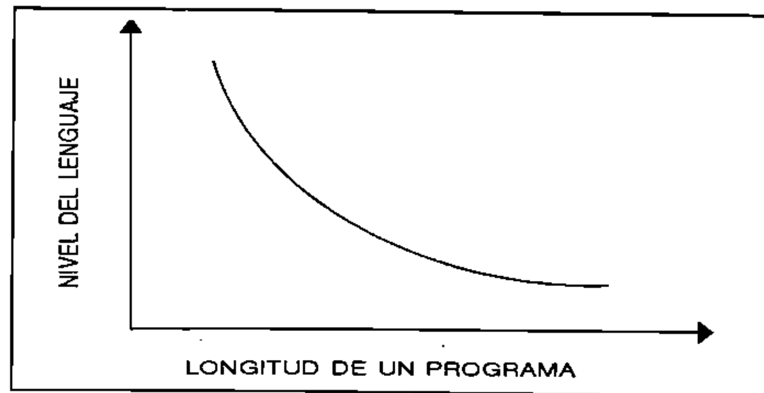


Fig. 6.2 Relación ideal entre λ y N.

Una última sugerencia para una investigación futura sería hacer el mismo análisis que se hizo en la presente tesis para otros 4GL's; como el PARADOX, PROGRESS y ORACLE.

APENDICE A

ANALIZADOR DE CODIGO FUENTE

Como se menciona en la sección 4.4, el analizador de código es la parte central de la investigación. Aunque el objetivo fundamental de la investigación fué probar ciertas métricas de Halstead para lenguajes de cuarta generación, no podemos negar la importancia fundamental que tiene el analizador en la investigación.

El analizador fué construído de tal forma que es capaz de hacer análisis de código para diferentes lenguajes haciendo sólo unas pequeñas modificaciones en el código fuente del analizador.

El analizador de código fue implementado en lenguaje PASCAL version 6.0. El disco que se anexa a la presente tesis contiene todos los programas en código fuente que hacen posible la utilización del analizador. Asimismo, si no se cuenta con un ambiente de PASCAL 6.0, el disco contiene el programa ejecutable del analizador.

<u>OBJETIVOS DEL ANALIZADOR.</u>

1).- Evitar errores humanos en la interpretación de las partes que componen el programa. Esto es, si una palabra es

operador o no.

2).- Evitar el conteo manual de los diferentes operadores y operandos que forman parte de el código fuente de un programa; es posible olvidar si una palabra es operador o no.

3).- Obtener las métricas de Halstead, para cualquier programa fuente.

PROBLEMAS CON LA CONSTRUCCION DEL ANALIZADOR.

Los problemas fundamentales en la construcción del analizador fueron los siguientes:

1).- Desconocer cuales palabras, dentro del programa, no tenían efecto en la ejecución del programa.

2).- Desconocer cuales palabras, dentro del programa, serían consideradas como operadores.

SOLUCION AL PRIMER PROBLEMA CON LA CONSTRUCCION DEL ANALIZADOR.

De acuerdo a las reglas de conteo de código fuente creadas por Software Productivity Research (Jones, 1991) y a las consideraciones de Halstead con respecto a las palabras que no tenían efecto en la ejecución de un programa, se llegó a la conclusión de que para los lenguajes DBASE III y FOXPRO2 las palabras que no tienen efecto en la ejecución de un programa son las que vienen escritas en el archivo de texto ARCH_BAS.TXT. Asimismo, se eliminaron las líneas de

comentarios. En los lenguajes DBASE y FOXPRO2, los comandos que nos indican los comentarios son: &&, NOTE, *.

SOLUCION AL SEGUNDO PROBLEMA CON LA CONSTRUCCION DEL ANALIZADOR.

De acuerdo a la definición de Halstead de operador, consideramos que el conjunto ideal de operadores que podrían venir en un programa es el conjunto formado, principalmente, por:

1).- Todos los posibles comandos que pudiera encontrar en el lenguaje a analizar.

2).- Todas las funciones que están implícitamente definidas en el lenguaje.

3).- Todos los operadores aritméticos, lógicos, y de puntuación definidos en el lenguaje.

Esta consideración establece la lista de posibles operadores del orden de billones (ver sección 5.1). Esto condujo a escribir los operadores en un archivo de texto para eliminar este problema. Los conjuntos de operadores para los lenguajes DBASE III y FOXPRO2 pueden verse en los archivos de texto ARCH_DB3.TXT y ARCH_FOX.TXT respectivamente, los cuales están grabados en el disco que viene anexo a la presente tesis. Los operandos son todas las palabras (o símbolos) que no son operadores.

RECOMENDACIONES CON RESPECTO AL ANALIZADOR.

Las personas que estén interesadas en utilizar el analizador de código sin entrar al ambiente de PASCAL 6.0 deben hacer lo siguiente:

1) Crear un directorio en el disco duro de su computadora con el nombre de ANALIZA.

2) Copiar el contenido del disco en el directorio ANALIZA que crearon en el disco duro de su computadora.

3) Dentro del directorio ANALIZA, teclear ANALIZA.

A las personas que estén interesadas en revisar el programa fuente para hacer mejoras al analizador de código (mejorar el archivo de palabras sin efecto en le ejecución, mejorar el archivo de operadores, hacer más elegante la presentación del sistema, etc.) se les recomienda que no hagan modificaciones directas en el programa fuente que viene en el disco. En vez de ésto, se les sugiere que copien los programas fuente a otro disco para que puedan hacer las modificaciones convenientes.

A las personas que quieran utilizar el analizador de código en otros lenguajes, tienen que tomar en cuenta los problemas asociados con la construcción del analizador de código. Esto es, tienen que definir primero cuáles son las palabras que no tienen efecto en la ejecución del programa, así como definir cuáles son los operadores dentro de un

programa escrito en el lenguaje que se va a analizar. En otras palabras, deben de construir dos archivos, uno equivalente a ARCH_BAS.TXT y otro al ARCH_FOX.TXT para su lenguaje particular. Asimismo, investigar cuales son los comandos que indican los comentarios en un programa fuente escrito en el lenguaje que se analiza. Por ejemplo, en PASCAL los comandos que indican comentarios son: {...TEXTO....}, (*...TEXTO...*).

Por último presentamos en Fig. A1 y A2, el diagrama de contexto y el diagrama de flujo de datos del analizador de código respectivamente. A continuación ofrecemos el significado de algunos de los nombres que intervienen en el diagrama de flujo de datos.

1).- ARCHIVO BASURA. Es el archivo que contiene las palabras que no tienen efecto en el conteo (basura).

2).- LIMPIA BASURA. Es el proceso que elimina la basura de un programa.

3).- MODIFICACION 1. Programa sin basura.

4).- BORRA COMENTARIOS. Es el proceso que elimina los comentarios, espacios en blanco y TABS, dentro de un programa.

5).- MODIFICACION 2. Programa sin comentarios, espacios en blanco y TABS.

6).- COMILLAS. Es el proceso que elimina los operandos que

vienen entre comillas dentro de un programa.

7).- MODIFICACION 3. Programa sin comillas.

8).- MODIFICACION 4. Programa sin operadores.

"Ojalá y que les sea útil el analizador de código"

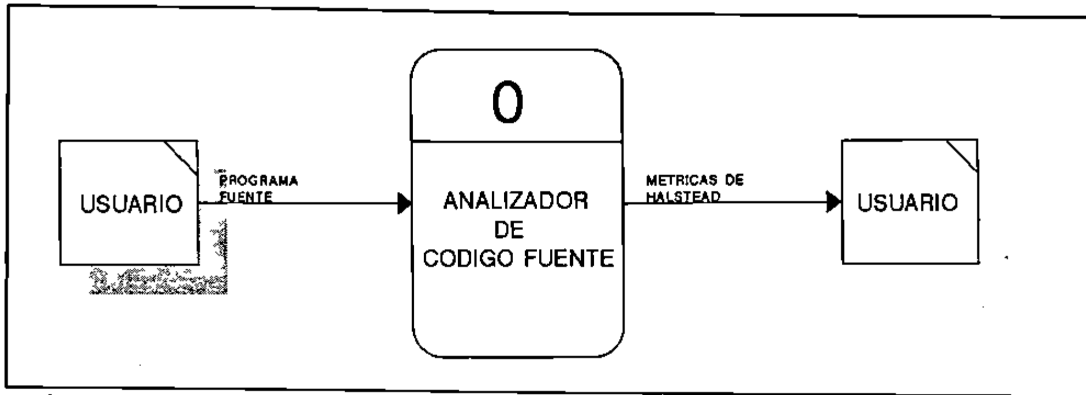


Fig. A.1 Diagrama de contexto del analizador de código.

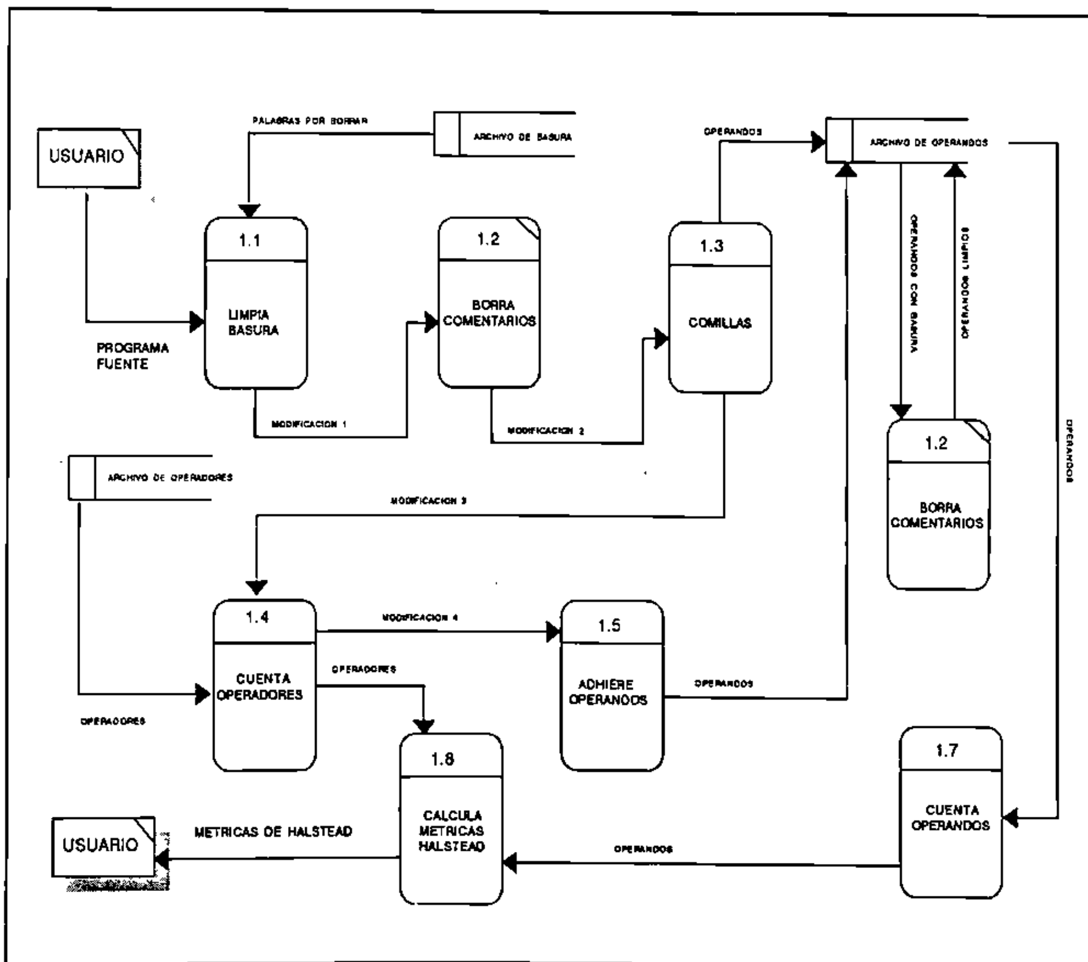


Fig. A.2 Diagrama de flujo de datos del analizador de código.

BIBLIOGRAFIA

Athey, T. H. y Zmud, R. W. Introduction to Computers and Information Systems, second edition; Scott, Foresman and Company, 1988.

Bowman, B. J. y Newman W. A. "Software Metrics as a Programming Training Tool", J. Systems Software, 1990, Vol. 13, pp. 139-147.

Crespo D. y Vigil R. "Diagnóstico de la situación actual de las unidades de informática en la mediana empresa de Cd. Victoria", Tesis de Licenciatura, Licenciado en Computación Administrativa, UAT, Noviembre de 1992.

Emory, C. W. y Cooper, D. R. Business Research Methods, fourth edition; Irwin, 1991.

Halstead, M. H. Elements of Software Science, Elsevier North-Holland, 1977.

Jones, C. Applied Software Measurement, Assuring Productivity and Quality, McGraw-Hill, 1991.

Kvanli, A., Guynes, C. y Pavur, R. Introduction to Business Statistics: A Computer Integrated Approach, second edition; West Publishing Company, 1989.

McCabe, T. "A Software Complexity Measure", IEEE Transaction Software Engineering, Vol. 2, Diciembre 1976, pp. 308-320.

McCall, J., Richards P., y Walters G., "Factors in Software Quality", General Electric, Command and Information Systems, Technical Report 77CIS02, Sunnyvale, California, 1977.

Pinter, L. Aplique Foxpro, McGraw-Hill/Interamericana de España, S. A., 1992.

Pressman, R. S. Ingeniería del Software un enfoque práctico, tercera edición; McGraw-Hill/Interamericana de España, S. A., 1993.

Ramamurthy B. y Melton, A. "A Synthesis of Software Science Measures and the Cyclomatic Number". IEEE Transactions on Software Engineering, Vol. 14, No. 8, Agosto 1988, pp. 1116-1121.

Shen, V. Y., Conte, S. D. y Dunsmore, H. E. "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support", IEEE Transactions on Software Engineering, Vol. SE-9, No. 2, March 1983, pp. 155-165.

Sigwart, Ch. D., Van Meer, G. L., y Hansen, J. C. Software Engineering a project oriented approach, Franklin, Beedle & Associates, 1990.

Weyuker, E. J. "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, Vol. 14, No. 9, September 1988, pp. 1357-1365.

Wrigley, C. D. y Dexter, A. S. "A model for Measuring Information System Size", MIS Quarterly, Junio 1991, pp. 245-257.

Yourdon, E. Decline & Fall of the American Programmer, Yourdon Press Computing Series, Prentice-Hall, 1992.