LA CALIDAD DEL SOFTWARE: UNA NECESIDAD

José L.Roca[*]
Ricardo O.Juliá[*]

En la mayoría de los proyectos actuales el hardware y el software comparten responsabilidades a la hora de las fallas. Más aun, el 80 por ciento del esfuerzo del desarrollo es puesto en el software mientras que el 20 por ciento restante corresponde al hardware.

Introducción

Hace tres décadas, cuando el grueso del diseño pasaba por el hardware específico para una cierta aplicación, la calidad y la confiabilidad del producto estaban centrados en técnicas conocidas y que actualmente siguen usándose, como realimentación, mejores requerimientos en cuanto a disipación de calor, mejor calidad intrínseca de componentes, redundancia, etcétera. En cuanto a la fabricación, el control de calidad por atributos o por variables con sus métricas: el AOQL (*Average Outgoing Quality Level*) como medida de la salida, la inspección de entrada con sus AQL (*Acceptable Quality Level*) y LTPD (*Low Total Percentage Defects*).

[*] Universidad Nacional de La Matanza.

Más cerca de nuestros días, el control estadístico de procesos (SQC) y el TQM (*Total Quality Management*) con sus distintas aplicaciones a procesos específicos de producción, componentes más confiables y mejores técnicas de fabricación de componentes básicos. Desde hace una década la influencia del software sobre un proyecto global es cada vez mayor. Tratándose de diseño de sistemas, el producto es un sistema pequeño o grande pero en él está subyacente la idea de un hardware de base sobre el cual corre un software, diseñado *ad hoc*.

Son pocos los que utilizan las herramientas propuestas por la denominada Ingeniería del software para mejorar la calidad y confiabilidad de un programa. Las aplicaciones pueden ser críticas o no, depende mucho del balance entre los conceptos disponibilidad-seguridad. Un sistema seguro nunca funcionará, mientras un sistema siempre disponible nunca será totalmente seguro. Un *Bug* (error) en un software puede provocar una falla que termine con una misión operativa aeroespacial; por cierto en esta industria y en la nuclear fueron donde nacieron las técnicas más usuales para lograr softwares más confiables. Esto sucedió mucho antes también con el hardware. Un error en la programación puede desencadenar una catástrofe en un sistema teleinformático o, más aun, puede implicar vidas humanas cuando el software es de aplicación electromédica.

Por otro lado, en lo que respecta a costos, el cambio en estas últimas tres décadas ha sido muy marcado. Mientras en los años 60 la relación de costos entre software y hardware era de 10-90 por ciento, esta relación es actualmente de 90-10 por ciento (véase figura 1). Esta es la importancia que día a día está teniendo el software en un sistema. El peso es considerable y justifica la obtención de softwares cada vez mejores a costos adecuados. Esto es una necesidad de mercado. La calidad hace al mercado. Es necesario, luego, producir softwares con niveles de calidad adecuados, con un bajo número de errores residuales, flexibles, portables y con altas cifras de confiabilidad. Para esto es necesario utilizar nuevas técnicas y modelos, nuevas métricas y un nuevo concepto en la ingeniería, la ingeniería del software.

Calidad y confiabilidad

Al hablar de estos dos términos a la hora del software es necesario clarificar el tema: mientras la calidad es una medida de la performance de un elemento en un punto determinado del tiempo, presumiblemente t=0, la confiabilidad es una medida de la performance a lo largo del tiempo. La performance de cualquier equipo o sistema puede ser medida en función de sus parámetros operativos a la salida de la línea de montaje y esta medida evalúa si están o no dentro de lo previsto por el fabricante. La repetibilidad de este experimento lo hace estadísticamente tratable. La confiabilidad tie-

ne un poco más que ver con la conservación de estos parámetros operativos a lo largo del tiempo y también, debido a las mismas causas que antes, la hacen probabilísticamente tratable. En otras palabras, una está relacionada con la variable aleatoria "proporción" o "porcentaje" y la otra con la variable aleatoria "tiempo".

El software no admite la misma discriminación. Desde su inicio hasta la liberación transcurre un tiempo y se puede hablar de calidad y confiabilidad de un software, pero en otros términos. La calidad expresada como proporción de errores (*Bugs*) en el programa, por ejemplo, mientras confiabilidad representa la tasa de fallas (*Hazard failure rate*). Son muchas las medidas utilizadas y muchas veces no se sabe si se trata de calidad o confiabilidad. Sin embargo en softwares, la mayoría de los especialistas hablan de confiabilidad y calidad en forma indistinta. Lo cierto es que un software no puede producirse en serie varias veces.

Una vez desarrollado, probado y liberado, todos los softwares serán copia de ése en cualquier sistema que se corran. Los errores, no así las fallas, serán siempre los mismos, si el entorno en el que se corren es el mismo. Es por ello que es mucho más correcto hablar de confiabilidad de un software que de su calidad.

La confiabilidad de un software

Como se aclaró en el punto anterior existe mucha discusión sobre la definición de la confiabilidad de un software. Por ejemplo, se dice que un software es confiable si realiza lo que el usuario desea cuando así lo requiera. No es confiable si así no lo hiciera. A nuestros fines un software no es confiable cuando falla. Las fallas se deben a errores en el software, así que si corregimos estos errores sin introducir nuevos estamos mejorando la confiabilidad del software.

En la figura 2 se presentan las posibles causas que originan errores. Se observa que aproximadamente el 72 por ciento de los errores se originan en "trasladar los requerimientos del usuario y el diseño lógico". Es así como podremos aumentar la confiabilidad de un software haciendo hincapié en estas primerísimas dos importantes etapas. Fuentes de diversos tipos aseveran que es en el diseño en donde debe ponerse énfasis para reducir la proporción de errores. En la figura 3 se muestra una apertura por tareas tipificadas. Aquí se observan nueve categorías en las que se ha dividido la generación de errores. La experiencia demuestra que aproximadamente el 76 por ciento de los errores no son descubiertos hasta bien entrada la etapa de pruebas integrales.

En la figura 4 se ilustra cuánto es el costo en veces de corrección de errores (*Debugging*). El costo relativo de la detección y corrección de errores durante y después de las etapas de integración y test pueden resultar

entre diez y quince veces mayor que en las etapas primeras de desarrollo y codificación. Estudios realizados concluyen que el ambiente donde se desarrolla el software contribuye enormemente al aumento de errores. La confiabilidad del software pasa a ser un problema de "management" y no "técnico".

Originalmente el software era considerado como algo misterioso y hasta esotérico, lleno de términos sofisticados y extraños que sólo podían ser descifrados por expertos en el tema. Los ingenieros de hardware no comprendían al mundo del software. El usuario entendía menos. Los líderes de proyectos delegaban, consecuentemente, responsabilidades en los supervisores del software para manejar esa parte del proyecto. A la hora de la revisión del diseño, cuando llegaba el momento de discutir sobre software, cliente y contratista normalmente se iban a otra sala para hablar de sus ceros, unos y bits. Todo este ambiente no estructurado resultaba en una total falta de control por parte del líder del proyecto. Ninguna metodología se había aún implementado al respecto. Los procesos de debugging eran escasos cuando no nulos. Los errores eran corregidos a medida que iban apareciendo sin ninguna metodología específica. Tampoco se utilizaba la realimentación como técnica para evitar errores en proyectos futuros. Los errores eran tan sólo documentados para asegurar su corrección y no eran disponibles sino para los intervinientes en el proyecto.

Los costos de desarrollo de softwares variaban entre 0,3 y 0,8 horas/hombre por palabra en memoria y ningún análisis era hecho para explicar el porqué de tan grande variabilidad.

Errores y fallas de un software

Históricamente, una forma de aumentar la confiabilidad de un software era correrlo y probarlo extensivamente antes de liberarlo. Los errores encontrados eran corregidos. La documentación asociada a la corrección era la descripción del error, momento de ocurrencia, parámetros operacionales en el momento de la ocurrencia y resolución del error. Tomando en cuenta las lecciones de los ingenieros de calidad en hardware, se puede decir que la confiabilidad deberá ser diseñada en el producto.

Teniendo un 72 por ciento de errores generados en el traslado de los requerimientos del usuario, el énfasis debe ser puesto en ese punto. Es mucho más efectivo resolver los errores en la misma fase de diseño que en la de prueba. Además, cada vez que se corrige un error se generan nuevos con una cierta probabilidad. Es mucho más costoso encontrar, corregir y documentar errores en los últimos peldaños del ciclo de vida que al comienzo.

Es pues necesario utilizar herramientas, que sobre la base de modelos, ayuden a determinar parámetros que sirvan de base de análisis. En la figura 5 se observa el proceso operativo de manejo de modelos de estimación de

performance. Estos modelos son variados y mucho dependen del estadio del ciclo de vida del software como también de las distintas aplicaciones. El modelo plantea una estrategia que se evalúa mediante prueba de ajuste o bondad. El resultado indicará si se deberá elegir otro modelo o perseverar en el mismo.

Pero, ¿por qué aparece una falla al correr un programa? En general un programa establece una correspondencia entre entradas y salidas a través de una determinada función estimada o propuesta. La diferencia entre esta función y la real ejecutada por el programa es lo que da origen a la falla. En realidad son las entradas las que interactúan con un determinado error en la programación y por lo tanto generan salidas no esperadas (fallas).

Por ejemplo, un sistema de tiempo real, en el cual del resultado de una determinada operación entre parámetros operacionales depende el futuro estado, ya sea vía actuadores o no. ¿Qué sucedería si uno de esos parámetros está en el denominador de esta operación y nunca se prueba en el programa si es cero? Es evidente que cuando lo sea, dará origen a una falla. El error es no probar por cero este operando y la falla un overflow y su posterior acción desencadenante sobre los actuadores del sistema.

Así es como errores provocan fallas y su corrección errores nuevos (véase figura 6). Un buen proceso de debugging debe seguir los pasos indicados en la figura 7, a los fines de prevenir el ingreso de errores nuevos y evitar la divergencia de los parámetros característicos del modelo seleccionado. En la misma figura 7 se muestran los errores típicos. Del análisis de fallas surge el aprendizaje para evitar futuros errores. Este vehículo asegura que en futuros emprendimientos no se cometan los mismos errores que en el pasado.

Modelos de confiabilidad de un software

En las figuras 8, 9 y 10 se muestran tres clasificaciones importantes de los modelos utilizados en el análisis de la confiabilidad de un software. La primera clasificación de acuerdo con los distintos momentos o fases del ciclo de vida, se debe a Ramamorthy y Bastani, de Berkeley, Universidad de California. Son realmente cuatro las fases, pero se ha agregado una última medida un poco descolgada del contexto de las distintas fases, debido a la importancia que en aplicaciones críticas adquiere. En la fase de desarrollo los modelos son de confiabilidad creciente (*growth reliability*). En la fase validación no se corrigen errores, sólo se prueba y se acepta o rechaza el software con un cierto grado de confianza. En la fase operacional la validación es continua, mientras que en la fase operacional se admite la adición de mejoras.

Otro punto de vista es el del proceso de la aparición de las fallas debida a Goel-Okumoto, de la Universidad de Syracuse. Los parámetros a medir

serán el tiempo medio entre fallas, el número de errores remanentes por vía del conteo de fallas, la reacción del programa ante la siembra de errores al azar y a la variabilidad y continuidad de las entradas.

En la tercera clasificación, debida a Shooman, del Instituto Politécnico de Brooklyn, se tiene en cuenta la estructura interna o no del software, dando origen a los macromodelos y a los micromodelos. Los modelos de disponibilidad cierran la clasificación poniendo en evidencia la interacción de los tiempos de corrección o debugging, los tiempos de aparición de fallas y la disponibilidad del software en la denominada fase de producción.

Obtención de softwares confiables

Mucho es lo que podría hacerse para incrementar la confiabilidad de un software si los que trabajan en el medio supiesen algunos simples y bien conocidos principios básicos. Muchos programadores meten textualmente programas en memoria pero poco hacen por ellos. Son pocos los grupos de trabajo sofisticados que hacen las cosas bien desde el principio. La mayoría escribe programas para resolver problemas específicos y no lo hacen bien.

Hay básicamente dos técnicas. La primera se basa en programación simple mientras la segunda utiliza elementos de *fault tolerant* (tolerancia a fallas). En la figura 11 se muestra esta clasificación.

Desde el punto de vista de programación simple el problema puede atacarse desde tres ángulos, a saber:

• **Estructuración**. Se puede comenzar comprendiendo el problema a resolver, dándole los primeros lineamientos que encaucen su resolución. Si se encuentra implementando un algoritmo escríbase top-down sus pasos más importantes. Así se tendrá la estructura gruesa de la resolución del problema. Algunas estructuras serán puramente secuenciales, otras más complejas. Trátese de darle "forma y color" a la estructura, hágasela simple con complejidades ciclomáticas que no superen diez. No deben admitirse "if" condicionales si no están "anillados" en jerarquías. Después de obtenida la estructura gruesa puede comenzarse con otras cosas más sofisticadas, como programación estructurada, si se lo desea.

• **Modularización**. Una vez que se tiene alguna estructura gruesa del programa, todo lo que literalmente "cuelga" de la estructura son los módulos. Cuídese que éstos no sean extensos pero suficientes para ejecutar paso a paso lo propuesto. Es muy conveniente que los módulos no superen una página de longitud, a lo sumo dos, debido fundamentalmente a factores humanos (fatiga de debugging).

• **KISS**. (*Keep it simple stupid*). Modularizar y estructurar ayuda a incrementar la confiabilidad de un software, pero no se debe complicar la

estructura aumentando la complejidad a costa de obtener una mejoría innecesaria en la performance.

Estas tres técnicas simples de programación deberían ser introducidas dentro del ambiente de ingeniería de software lentamente, sin prisa pero sin pausa, gradualmente. No se trata de revolucionar la técnica sino el management. Hágase evolucionar el trabajo con mejores estructuras, mejores módulos y siempre KISS.

Cuando se necesitan obtener altas cifras de confiabilidad, las técnicas precedentes no bastan y es necesario recurrir a técnicas de programación más complejas como las de tolerancia a fallas. Actualmente se utilizan dos, a saber:

• **N-versiones de un programa** (*N-version programming*). Se trata de N versiones de un programa corriendo simultáneamente con N>2. Estas versiones son codificadas en forma independiente y los resultados son comparados al final, verificándose o no su concordancia. En caso de no concordar se tomará una determinada decisión de acuerdo con una determinada estrategia. Esta técnica requiere N hardwares para correr las N versiones del programa y su uso está restringido a casos especiales.

• **Bloques recuperables** (*Recovery blocks*). Se reúne a varias rutinas alternativas enganchadas a través de una prueba o condición que se espera como resultado positivo de cada una de ellas. Un control interno realiza la transferencia de una rutina a otra cuando la condición impuesta por la prueba no se cumple. Asimismo, un control de tiempo (*watchdog*) monitorea los tiempos de ejecución de ambas. Esta técnica es usada ampliamente pues requiere un solo hardware.

Conclusión

Obtener softwares cada vez más confiables es, tanto desde el punto de vista práctico como ético, necesario. Los objetivos sólo pueden alcanzarse mediante la aplicación sistemática de herramientas, a veces poco conocidas. Sin embargo, la importancia que el software está teniendo dentro de un sistema justifican, ya sea por costos o por performance, su utilización.

La divulgación de este paquete de conocimiento debe comenzar desde los primeros pasos de la enseñanza universitaria y propagarse a lo largo de cada emprendimiento que en materia software se comience. Con mejores softwares, menos complejos y más portables se podrán obtener mejores resultados aplicativos. Las herramientas están, sólo hay que utilizarlas.

BIBLIOGRAFIA

ADELSON, B. Y SOLOWAY, E., "The Role of Domain Experience in Software Design", IEEE Transactions on Software Engineering, Vol.SE-11, N°11, noviembre de 1985.

BOHEM, B.W., "Software Engineering Economics", Prentice Hall Inc.,Englewood Cliffs, NJ 07632, 1981.

BRACKER LYNNE, C., "Software Reliability",12th Annual Reliability Testing Institute, abril de 1986, Tucson, Arizona, USA.

BRITCHER, R., "Using Inspections to Investigate Program Correctness", IEEE Computer, noviembre de 1988.

BUKOWSKI, J. Y GOBLE, W., "Practical Lessons for Improving Software Quality", 1990 Proceedings Annual Reliability & Maintainability Symposium IEEE.

ENDRES A., "An Analysis of Errors and their Causes in Systems Programs", IEEE Transactions on Software Engineering, junio de 1975, pp.140-149.

EVANS, R.A. Y REGULINSKI, T.L., "Right, Here & Now", IEEE Transactions on Reliability, Vol.R-28, N°3, agosto de 1979.

EVANS, R.A. Y REGULINSKI, T.L., "On Software Reliability Models", IEEE Transactions on Reliability, Vol.R-28, N°3, agosto de 1979.

HECHT, H., "Fault Tolerant Software", IEEE Transactions on Reliability, Vol.R-28, N°3, agosto de 1979, pp.229-232.

MUSA, J.D., "Software Reliability", McGraw Hill, 1987.

PARNAS, D. Y WEISS, M., "Active Design Reviews: Principles & Practices", Proceedings of the 8th International Conference of Software Engineering, IEEE Computer Society Press, 1985.

YATES III, W. Y SHALLER, D.A., "Reliability Engineering as Applied to Software", 1990 Proceedings Annual Reliability & Maintainability Symposium IEEE.

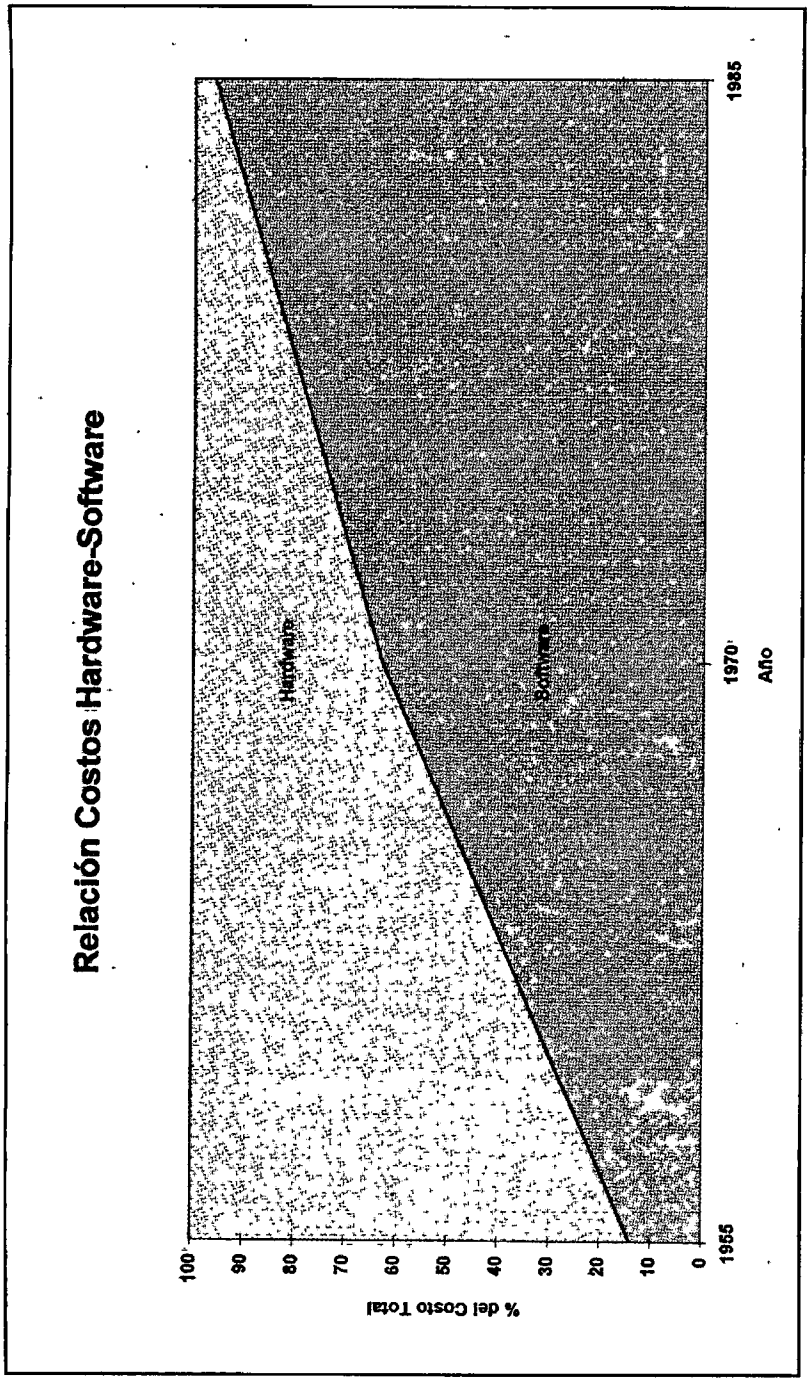

Relación Costos Hardware-Software
Hardware
Software
% del Costo Total
100
90
80
70
60
50
40
30
20
10
0
1955
1970
1985
Año

Fig.1

Proporción de Errores de un Software durante su Ciclo de Desarrollo

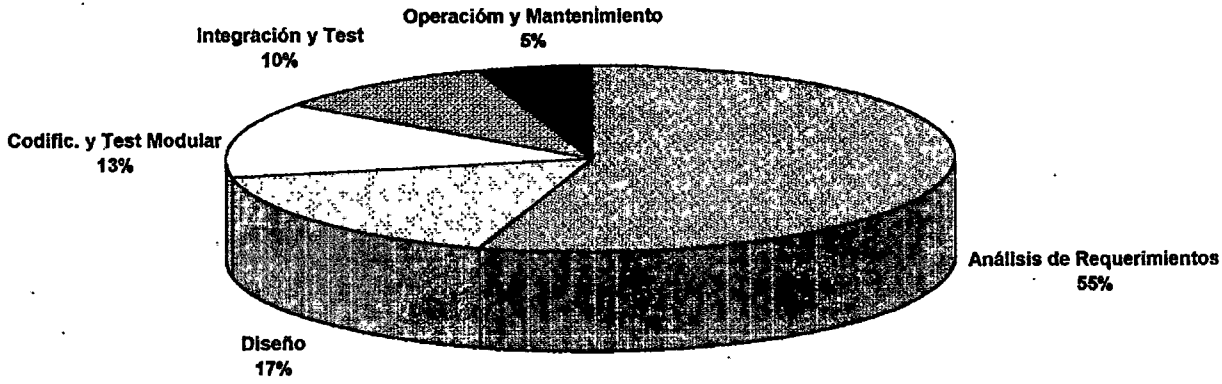


Fig.2

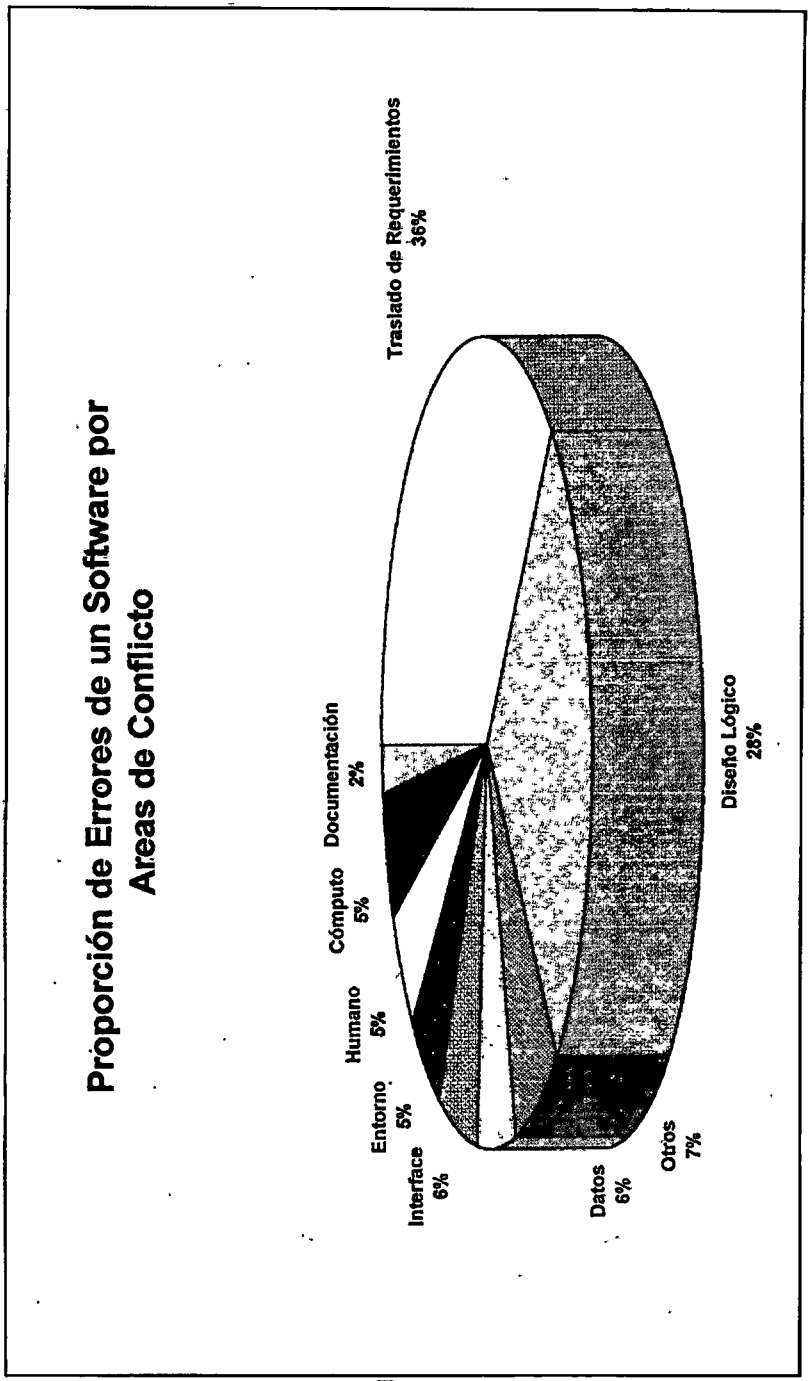

Proporción de Errores de un Software por Areas de Conflicto
Traslado de Requerimientos 36%
Diseño Lógico 28%
Otros 7%
Datos 6%
Interface 6%
Entorno 5%
Humano 5%
Cómputo 5%
Documentación 2%

Fig.3

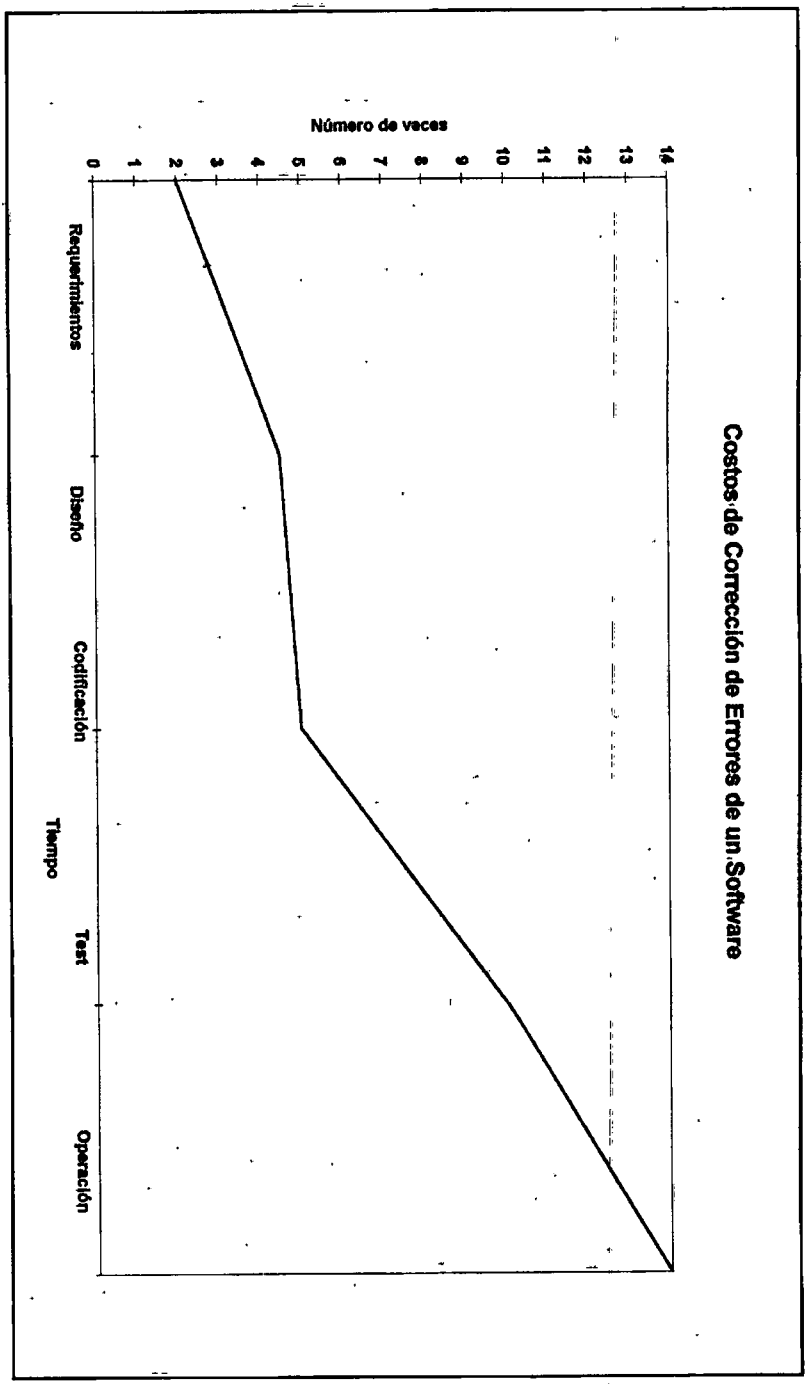

Costos de Corrección de Errores de un Software
Número de veces
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Requerimientos
Diseño
Codificación
Test
Operación
Tiempo

Fig.4

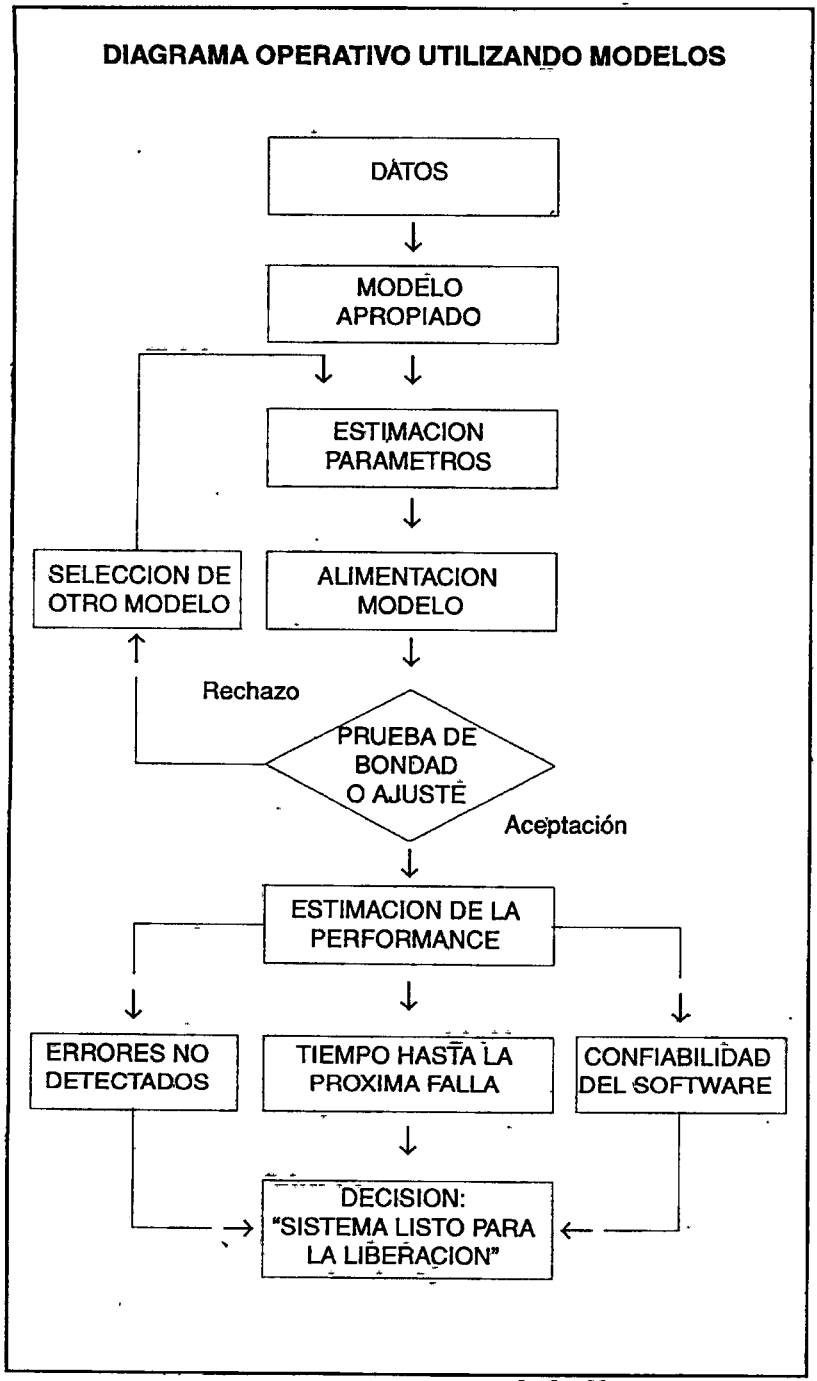
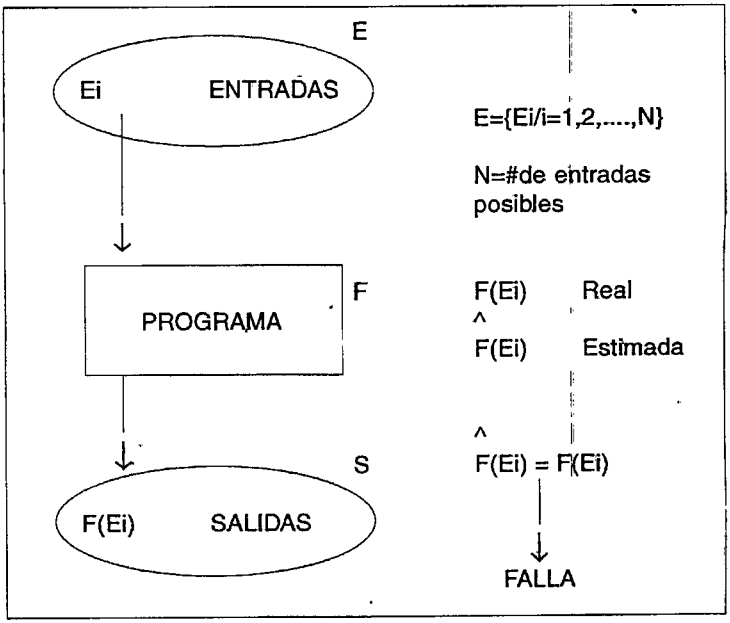


DIAGRAMA OPERATIVO UTILIZANDO MODELOS
DATOS
MODELO APROPIADO
ESTIMACION PARAMETROS
SELECCION DE OTRO MODELO
ALIMENTACION MODELO
Rechazo
PRUEBA DE BONDAD O AJUSTE
Aceptación
ESTIMACION DE LA PERFORMANCE
ERRORES NO DETECTADOS
TIEMPO HASTA LA PROXIMA FALLA
CONFIABILIDAD DEL SOFTWARE
DECISION: "SISTEMA LISTO PARA LA LIBERACION"

Fig.5

ERRORES Y FALLAS DE UN SOFTWARE



¿Por qué falla un software?
Ignorancia de los requerimientos del ususario
Ignorancia del entorno en que se utiliza el software
Escaso flujo de información entre programador y usuario
Escasa documentación del software

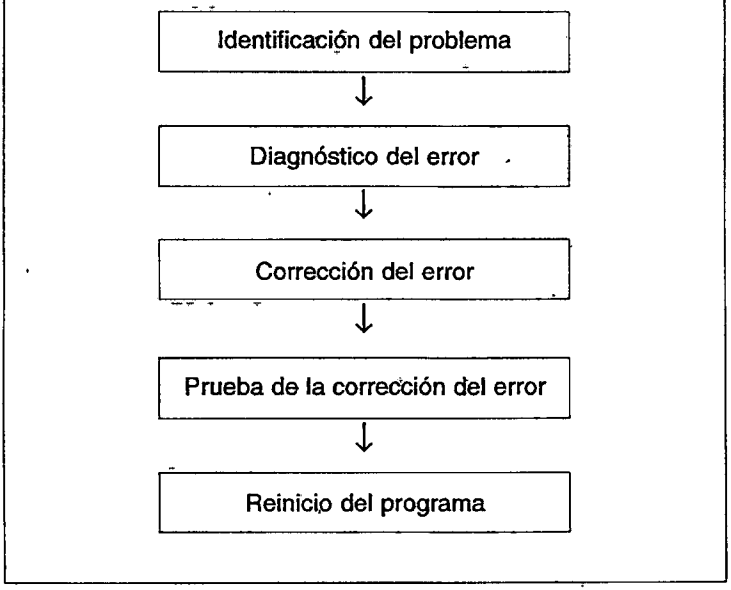
ERRORES (BUGS) ——— → FALLAS

Errores previos fijos: persisten en el software luego de que el programador ha trabajado en él corrigiendo un error o cambiando un código (Debugging).

Errores generados: no existían en el software, hasta que son introducidos como consecuencia del Debugging.

Fig.6

PROCESO DE DEBUGGING



ERRORES CLASICOS

Matrices sobregrabadas
Errores en los bits de bandera
Errores en los bits de indexado
Errores en los bits de desplazamiento
Errores de complementación aritmética
Problemas de puntero
Errores de transferencia de control
Problemas de direccionamiento indirecto
Problema de reinicio del programa

Fig.7

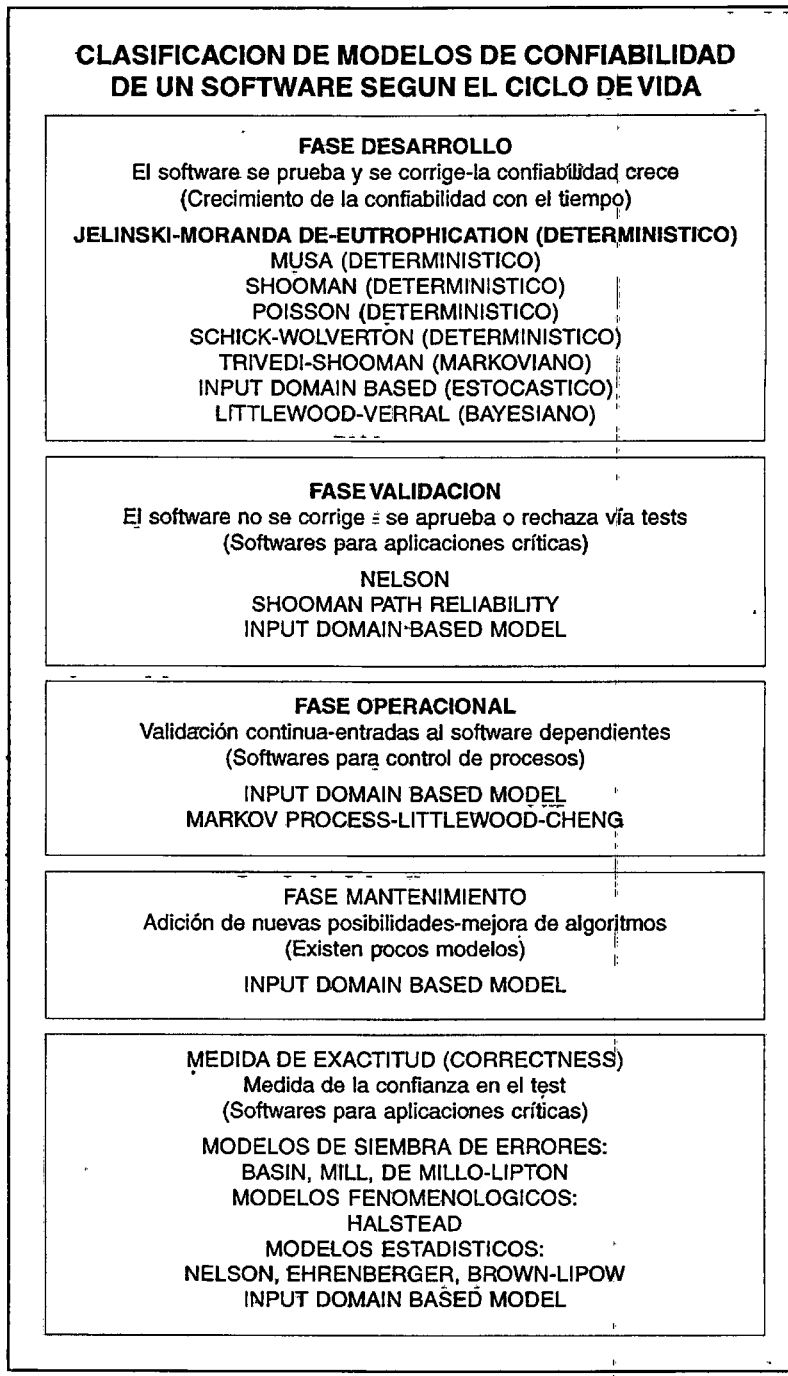

CLASIFICACION DE MODELOS DE CONFIABILIDAD DE UN SOFTWARE SEGUN EL CICLO DE VIDA
FASE DESARROLLO
El software se prueba y se corrige-la confiabilidad crece
(Crecimiento de la confiabilidad con el tiempo)
JELINSKI-MORANDA DE-EUTROPHICATION (DETERMINISTICO)
MUSA (DETERMINISTICO)
SHOOMAN (DETERMINISTICO)
POISSON (DETERMINISTICO)
SCHICK-WOLVERTON (DETERMINISTICO)
TRIVEDI-SHOOMAN (MARKOVIANO)
INPUT DOMAIN BASED (ESTOCASTICO)
LITTLEWOOD-VERRAL (BAYESIANO)
FASE VALIDACION
El software no se corrige = se aprueba o rechaza vía tests
(Softwares para aplicaciones críticas)
NELSON
SHOOMAN PATH RELIABILITY
INPUT DOMAIN-BASED MODEL
FASE OPERACIONAL
Validación continua-entradas al software dependientes
(Softwares para control de procesos)
INPUT DOMAIN BASED MODEL
MARKOV PROCESS-LITTLEWOOD-CHENG
FASE MANTENIMIENTO
Adición de nuevas posibilidades-mejora de algoritmos
(Existen pocos modelos)
INPUT DOMAIN BASED MODEL
MEDIDA DE EXACTITUD (CORRECTNESS)
Medida de la confianza en el test
(Softwares para aplicaciones críticas)
MODELOS DE SIEMBRA DE ERRORES:
BASIN, MILL, DE MILLO-LIPTON
MODELOS FENOMENOLOGICOS:
HALSTEAD
MODELOS ESTADISTICOS:
NELSON, EHRENBERGER, BROWN-LIPOW
INPUT DOMAIN BASED MODEL

Fig.8

CLASIFICACION DE MODELOS DE CONFIABILIDAD DE UN SOFTWARE SEGUN LA NATURALEZA DEL PROCESO DE FALLA

TIEMPO ENTRE FALLAS
Se estudia el tiempo entre fallas

JELINSKI-MORANDA DE-EUTROPHICATION
GOEL-OKUMOTO-IMPERFECTO DEBUGGING
SCHICK-WOLVERTON
LITTLEWOOD-VERRAL-BAYESIANO

CONTEO DE FALLAS
Se estudia el número de fallas detectadas

GOEL-OKUMOTO-POISSON NO HOMOGENEO
GOEL-POISSON NO HOMOGENEO GENERALIZADO
MUSA-TIEMPO DE EJECUCION
SHOOMAN-EXPONENCIAL
POISSON GENERALIZADO
IBM-BINOMIAL-POISSON
MUSA-OKUMOTO-POISSON LOGARITMICO

SIEMBRA DE ERRORES
Se estudia la reacción ante la introducción forzada de errores

MILLS

DOMINIO DE LAS ENTRADAS
Se estudia la reacción ante la variabilidad de entradas

NELSON
RAMAMOORTHY-BASTANI

MEDIDA DE EXACTITUD (CORRECTNESS)
Medida de la confianza en el test
(Softwares para aplicaciones críticas)

MODELOS DE SIEMBRA DE ERRORES:
BASIN, MILL, DE MILLO-LIPTON
MODELOS FENOMENOLOGICOS:
HALSTEAD
MODELOS ESTADISTICOS:
NELSON, EHRENBERGER, BROWN-LIPOW
INPUT DOMAIN BASED MODEL

Fig.9

CLASIFICACION DE MODELOS DE CONFIABILIDAD DE UN SOFTWARE SEGUN CONSIDERACIONES ESTRUCTURALES

MACROMODELOS
Se estudia el software como una caja negra

JELINSKI-MORANDA
GOEL-OKUMOTO
SCHICK-WOLVERTON
LITTLEWOOD-VERRAL
SHOOMAN
MUSA
NELSON
MILLS

MICROMODELOS
Se estudia la estructura interna del software

SHOOMAN

MODELOS DE DISPONIBILIDAD
Se estudia el proceso del mantenimiento del software

SHOOMAN-TRIVEDI

Fig.10

TECNICAS PARA INCREMENTAR LA CONFIABILIDAD DE UN SOFTWARE

TECNICAS DE PROGRAMACION SIMPLES
ESTRUCTURACION
MODULARIZACION
KISS

TECNICAS DE PROGRAMACION COMPLEJAS
N-VERSIONES DE UN PROGRAMA
BLOQUES RECUPERABLES

Fig.11