VERSIDAD NACIONAL
AVFNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN

CALIDAD EN LAS ORGANIZACIONES (EMPRESAS E INSTITUCIONES).

CALIDAD E INGENIERÍA DEL SOFTWARE ASISTIDA POR COMPUTADORA (CASE)

TRABAJO DE SEMINARIO

QUE PARA OBTENER EL TÍTULO DE :

INGENIERO MECÁNICO ELECTRICISTA

P R E S E N T A

ADOLFO VÁZQUEZ JUÁREZ

ASESOR : ING. JUAN DE LA CRUZ HERNÁNDEZ Z.

CUAUTITLAN IZCALLI, EDO. MEX. 1998.

VNIVERMDAD NACIONAL
AVTONOMA DE
MEXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLAN
UNIDAD DE LA ADMINISTRACION ESCOLAR
DEPARTAMENTO DE EXAMENES PROFESIONALES

U. N. A. M.
FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLAN

DEPARTAMENTO DE
EXAMENES PROFESIONALES

DR. JUAN ANTONIO MONTARAZ CRESPO
DIRECTOR DE LA FES-CUAUTITLAN
P R E S E N T E .

AT'N: Q. MA. DEL CARMEN GARCIA MIJARES
Jefe del Departamento de Exámenes
Profesionales de la FES-C.

Con base en el art. 51 del Reglamento de Exámenes Profesionales de la FES-Cuautitlán, nos permitimos comunicar a usted que revisamos el Trabajo de Seminario:

Calidad en las organizaciones (Empresas e Instituciones).
Calidad e Ingeniería del Software Asistida por Computadora (CASE).

que presenta el pasante: Adolfo Vázquez Juárez,
con número de cuenta: 8625550-3 **para obtener el Título de:**
Ingeniero Mecánico Electricista.

Considerando que dicho trabajo reúne los requisitos necesarios para ser discutido en el EXAMEN PROFESIONAL correspondiente, otorgamos nuestro VISTO BUENO.

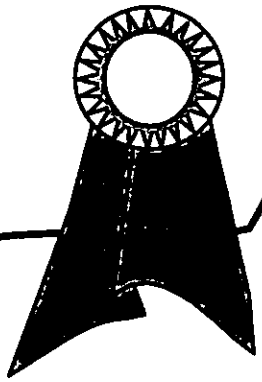
A T E N T A M E N T E .
"POR MI RAZA HABLARA EL ESPIRITU"
Cuautitlán Izcalli, Edo. de México, a 22 **de** Junio **de 19** 98

MODULO:	PROFESOR:	FIRMA:
I y III	Ing. Juan de la Cruz Hernández Zamudio	
II	Ing. Juan Rafael Garibay Bermudez	
IV	Ing. Jorge de la Cruz Trejo	

A mis Padres...
con todo mi corazón y agradecimiento
por su esfuerzo, apoyo y amor.

A H.D.R.
porque dejas en mí un rasgo de conciencia
de lo bueno que pueden ser las personas y
poder confiar en ellas.

A Wendy
por ser un gran motivo de superación en la vida.



C O N T E N I D O

PÁG.

▸ *INTRODUCCIÓN*

La idea de construir instrumentos que ayuden o faciliten al ser humano la realización de alguna tarea es común a todas las ingenierías. Por ejemplo, la Ingeniería Industrial construye, entre otras cosas, grúas que sustituyen al hombre en la tarea de levantar pesos; la Ingeniería Civil construye vías y puentes que facilitan al ser humano su tarea de trasladarse por tierra; la Ingeniería Naval construye barcos que ayudan al ser humano en la tarea de trasladarse por mar; la Ingeniería de Telecomunicaciones construye instrumentos que ayudan al hombre en la tarea de comunicarse, etc.

Una diferencia fundamental entre los instrumentos construidos por la Informática y los del resto de las ingenierías, es que estos últimos ayudan o sustituyen al ser humano en la realización de tareas físicas y aquéllos no. Cuando no existía la grúa, ni la palanca, el ser humano trasladaba pesos usando sus brazos; o antes de construir caminos, el ser humano caminaba campo a través; o, previamente a la existencia de barcos, el ser humano nadaba para trasladarse por el agua; o incluso antes de la existencia de las telecomunicaciones, inicialmente, el ser humano se comunicaba en largas distancias gritando, o, posteriormente, con señales luminosas o sonoras o de humo, etc.

Dicho con otras palabras, la Ingeniería Civil y Naval, construyen instrumentos que facilitan la realización de una tarea física (caminar y nadar); la Ingeniería Industrial y de Telecomunicaciones construyen instrumentos que prolongan o aumentan capacidades físicas del ser humano (levantar peso, por ejemplo, y comunicarse); lo mismo ocurre con las otras ingenierías.
Una mención especial merece la Ingeniería Aeronáutica, que ha dotado al ser humano de una capacidad física que no poseía: trasladarse a través del aire. En cualquier caso; todos estos instrumentos imitan, aumentan, ayudan, facilitan o sustituyen capacidades físicas del ser humano.

La Informática, sin embargo, se diferencia de todas ellas en que construye instrumentos que imitan, aumentan, ayudan, facilitan o sustituyen tareas psíquicas del ser humano. Inicialmente, imitaron la capacidad de cálculo matemático, pero, rápidamente, superaron y sustituyeron a las personas en la realización de esta tarea. A partir de ahí, se han construido instrumentos informáticos que han imitado capacidades psíquicas del ser humano cada vez más complejas: cálculo, almacenamiento y manejo de datos, realización de algoritmos, almacenamiento y manejo de información, y, en los últimos tiempos, ejecución de tareas consideradas inteligentes.

COMPONENTE HARDWARE Y COMPONENTE SOFTWARE

Para la imitación de tales capacidades «psíquicas» los sistemas informáticos, están compuestos de dos partes fundamentales: el hardware (componente físico y tangible: la computadora) y el software (componente lógico e intangible: los programas), Nótese que ocurre lo mismo con el miembro del cuerpo que dota al ser humano de la capacidad que las computadoras pretenden imitar: la resolución de problemas. El hardware humano para la resolución de problemas es el cerebro, mientras que su software es la mente, el pensamiento y la capacidad de inferencia y raciocinio.

La existencia de un componente hardware y un componente software también marca una diferencia fundamental entre la Informática y las otras ingenierías. Los instrumentos que, originalmente (sin intervención de la Informática), construyen las otras ingenierías, son instrumentos sólo hardware, de modo semejante a los miembros del cuerpo (manos, pies, boca, oído, etc.) que capacitan al ser humano de las tareas que pretenden imitar.

Esta diferenciación clara, entre hardware y ,software, de las partes de los instrumentos o sistemas informáticos lleva a una división también en el estudio de la Informática: el estudio de la construcción de hardware y el estudio del desarrollo y construcción de software.

▸ CAPITULO I : SOFTWARE E INGENIERÍA DEL SOFTWARE

♦ EL SOFTWARE

Hace veinte años, menos del 1 % de la gente podía describir de forma inteligente lo que significaba el "software de computadora". Hoy, la mayoría de los profesionales y muchas personas en general creen que entienden el software.
Pero, ¿realmente lo entienden?
Una descripción del software de un libro de texto puede tener la siguiente forma: "Software: (1) instrucciones (programas de computadora) que cuando se ejecutan proporcionan la función y el comportamiento deseado, (2) estructuras de datos que facilitan a los programas manipular adecuadamente la información, y (3) documentos que describen la operación y el uso de los programas". No hay duda de que podrían ofrecerse otras definiciones más completas, pero nosotros necesitamos algo más que una definición formal.

Características del software

Para poder comprender lo que es el software (y consecuentemente la ingeniería del software), es importante examinar las características del software que lo diferencian de otras cosas que los hombres pueden construir. Cuando se construye hardware, el proceso creativo humano (análisis, diseño, construcción, prueba) se traduce finalmente en una forma física. Si construimos una nueva computadora. nuestro boceto inicial, diagramas formales de diseño y prototipo de prueba, evolucionan hacia un producto físico (pastillas de VLSI, tarjetas de circuitos impresos, fuentes de potencia, etc.).

El software es un elemento del sistema que es lógico, en lugar de físico. Por tanto, el software tiene unas características considerablemente distintas a las del hardware :

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

Aunque existen algunas similitudes entre el desarrollo del software y la construcción del hardware, ambas actividades son fundamentalmente diferentes. En ambas actividades la buena calidad se adquiere mediante un buen diseño, pero la fase de construcción del hardware puede introducir problemas de calidad que no existen (o son fácilmente corregibles) en el software. Ambas actividades dependen de las personas, pero la relación entre la gente dedicada y el trabajo realizado es completamente diferente para el software. Ambas actividades requieren la construcción de un "producto", pero los métodos son diferentes.

Los costes del software se encuentran en la ingeniería. Esto significa que los proyectos de software no se pueden gestionar como si fueran proyectos de fabricación.

En la pasada década se ha tratado en la literatura el concepto de "fábrica de software" (p. ej.: [MAN84], [TAJ84]). Es importante tener en cuenta que este término no implica que la fabricación del hardware y el desarrollo del software sean equivalentes. En vez de ello, el concepto de fábrica de software recomienda el uso de herramientas para el desarrollo automático del software.

2. El software no se "estropea".

La Figura 1.1 describe, para el hardware, la proporción de fallos como una función del tiempo. Esa relación, denominada frecuentemente "curva de bañera", indica que el hardware exhibe relativamente muchos fallos al principio de su vida (estos fallos son atribuibles normalmente a defectos del diseño o de la fabricación) una vez corregidos los defectos, la tasa de fallos cae hasta un nivel estacionario (bastante bajo, con un poco de optimismo) donde permanece durante un cierto período de tiempo. Sin embargo, conforme pasa el tiempo, los fallos vuelven a presentarse a medida que los componentes del hardware sufren los efectos acumulativos de la suciedad, la vibración, los malos tratos, las temperaturas extremas y muchos otros males externos. Sencillamente, el hardware comienzo a estropearse.

El software no es susceptible a los males del entorno que hacen que el hardware se estropee. Por tanto, en teoría, la curva de fallos para el software tendría la forma que muestra la Figura 1.2 Los defectos no detectados harán que falle el programa durante las primeras etapas de su vida. Sin embargo, una vez que se corrigen, suponiendo que no se introducen nuevos errores la curva se aplana, como muestra la figura. Esa Figura es una gran simplificación de los modelos reales de fallos del software.

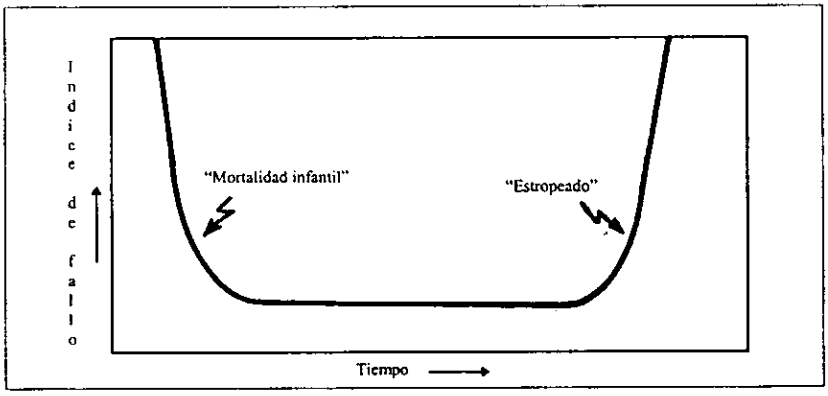


Figura 1 .1 Curva de fallos del hardware.

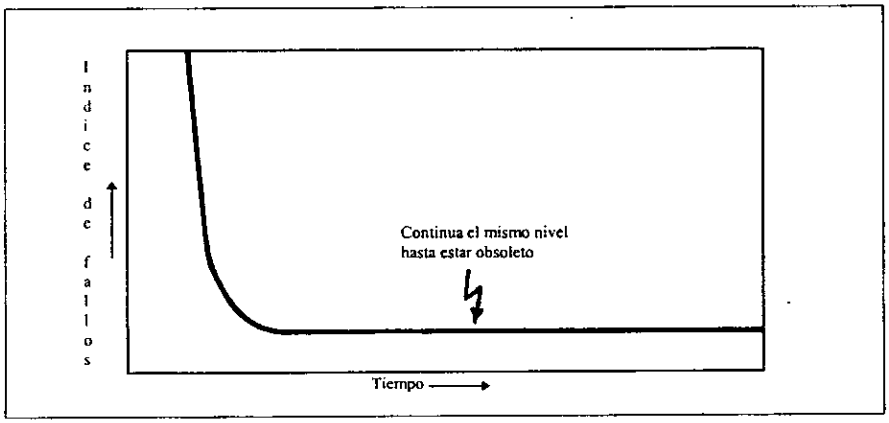


Figura 1 .2 Curva de fallos del software (idealizada).

Sin embargo, la implicación es clara el software no se estropea, pero se deteriora. Esto, que parece una contradicción, puede comprenderse mejor considerando la figura 1.3. Durante su vida, el software sufre cambios (mantenimiento). Conforme se hacen los cambios, es bastante probable que se introduzcan nuevos defectos, haciendo que la curva de fallos tenga picos como se ve en la figura .

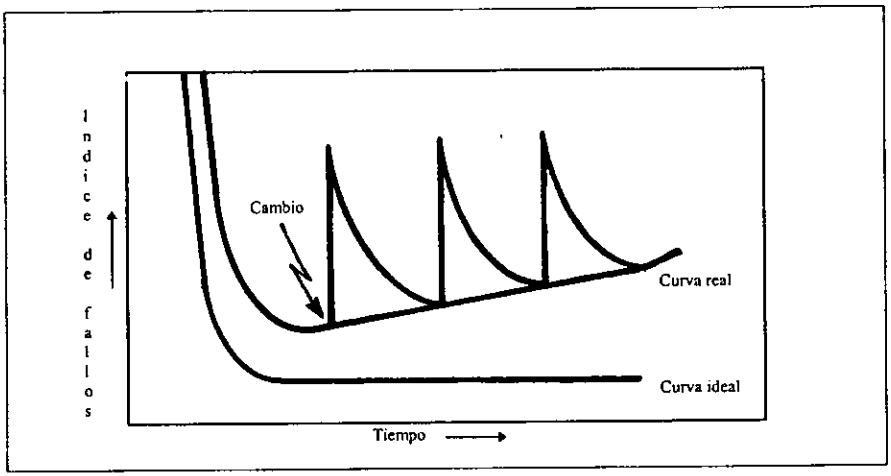


Figura 1.3 Curva real de fallos del software.

Antes de que la curva pueda volver al estado estacionario original, se solicita otro cambio, haciendo que de nuevo se cree otro pico. Lentamente, el nivel mínimo de fallos comienza a crecer; el software se va deteriorando debido a los cambios.

Otro aspecto de ese deterioro ilustra la diferencia entre el hardware y el software. Cuando un componente de hardware se estropea. se sustituye por una "pieza de repuesto". No hay piezas de repuesto para el software. Cada fallo en el software indica un error en el diseño o en el proceso mediante el que se tradujo el diseño a código máquina ejecutable. Por tanto, el mantenimiento del software tiene una complejidad considerablemente mayor que la del mantenimiento del hardware.

3. *La mayoría del software se construye a medida en vez de ensamblar componentes existentes.*

Consideremos la forma en la que se diseña y se construye el hardware de control para un producto basado en microprocesador. El ingeniero de diseño construye un sencillo esquema de la circuitería digital, hace algún análisis fundamental para asegurar que se realiza la función adecuada y va al catálogo de ventas de componentes digitales existentes. Cada circuito integrado (frecuentemente llamado un "CI" o "pastilla") tiene un número de pieza, una función definida y válida, una interfaz bien definida y un conjunto estándar de criterios de integración. Después de seleccionar cada componente, puede solicitarse la compra.

Por desgracia, los diseñadores de software no disponen de esa comodidad que acabamos de describir. Con unas pocas excepciones, no existen catálogos de componentes de software. Se puede comprar software ya desarrollado, pero sólo como una unidad completa, no como componentes que puedan reensamblarse en nuevos programas. Aunque se ha escrito mucho sobre "reusabilidad del software" (por ejemplo, [TRA88]), sólo estamos comenzando a ver las primeras implementaciones con éxito de este concepto.

Componentes del software

El software de computadora es información que existe en dos formas básicas : componentes no ejecutables en la máquina y componentes ejecutables en la máquina. Por lo que a este capítulo respecta, sólo contemplaremos los componentes de software que conducen directamente a instrucciones máquina ejecutables.

Los componentes de software se crean mediante una serie de traducciones que hacen corresponder los requisitos del cliente con un código ejecutable en la máquina. Se traduce un modelo (prototipo) de requisitos a un diseño.

Se traduce el diseño del software a una forma en un lenguaje que especifica las estructuras de datos, los atributos procedimentales y los requisitos que atañen al software. La forma en lenguaje es procesada por un traductor que la convierte en instrucciones ejecutables en la máquina.

La reusabilidad es una característica importante para un componente de software de alta calidad [BIG84]. Es decir, el componente debe diseñarse e implementarse para que pueda volver a usarse en muchos programas diferentes. En los años sesenta se construyeron bibliotecas de subrutinas científicas reutilizables en una amplia serie de aplicaciones científicas y de ingeniería. Esas bibliotecas de subrutinas reutilizaban de forma efectiva algoritmos bien definidos, pero tenían un dominio de aplicación limitado. Hoy en día, hemos extendido nuestra visión de reusabilidad para abarcar no sólo los algoritmos, sino también estructuras de datos. Un componente reusable de los años noventa encapsula tanto datos como procesos en un único paquete (frecuentemente llamado *(clase u objeto)*, permitiendo al ingeniero de software crear nuevas aplicaciones a partir de trozos reusables. Por ejemplo, las interfaces interactivas de hoy en día se construyen frecuentemente a partir de componentes reusables que permiten la creación de ventanas gráficas, de menús emergentes y de una amplia variedad de mecanismos de interacción. Las estructuras de datos y los detalles de procesamiento requeridos para construir la interfaz están contenidos en una biblioteca de componentes reusables orientados a la construcción de interfaces.

Los componentes de software se construyen mediante un lenguaje de programación que tiene un vocabulario limitado, una gramática definida explícitamente y reglas bien formadas de sintaxis y semántica. Estos atributos son esenciales para la traducción por la máquina. Las clases de lenguajes que se utilizan actualmente son los lenguajes máquina, los lenguajes de alto nivel y los lenguajes no procedimentales.

Los lenguajes de máquina son una representación simbólica del conjunto de instrucciones de la CPU. Si un buen programador produce programas mantenibles y bien documentados, puede utilizar el lenguaje máquina para hacer un uso extremadamente eficiente de la memoria y para "optimizar" la velocidad de ejecución del programa. Si el programa está mal diseñado y tiene poca documentación, el lenguaje máquina puede convertirse en una pesadilla.

Los lenguajes de alto nivel 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. De hecho, los compiladores e intérpretes de los lenguajes de alto nivel producen lenguaje máquina como salida.

Aunque hoy se utilizan cientos de lenguajes de programación, poco más de una decena son lenguajes de programación de alto nivel con una gran aceptación en la industria. Después de casi treinta años desde su aparición, lenguajes como COBOL y FORTRAN siguen utilizándose mucho en la actualidad. Los lenguajes de programación modernos (lenguajes que soportan directamente las prácticas modernas para el diseño procedimental y de datos) tales como Pascal, C y Ada se utilizan ampliamente. Los lenguajes orientados a los objetos como C++, Object Pascal, Eiffel y otros, están ganando cada vez más seguidores entusiastas.

Los lenguajes especializados (diseñados para ámbitos de aplicación específicos), como APL, LISP, OPS5 y lenguajes descriptivos para redes neuronales artificiales, están teniendo cada vez mayor aceptación conforme las nuevas aplicaciones pasan de los laboratorios a la práctica.

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 computadora.

Con cualquiera de esos lenguajes, el programador ha de preocuparse tanto de la especificación de la estructura de la información como de la de control del propio programa. Por ello, los lenguajes de las tres primeras generaciones se denominan *lenguajes procedimentales.*

En la década pasada, apareció un grupo de lenguajes *no procedimentales* o de cuarta generación. En vez de requerir que quien desarrolla el software especifique los detalles procedimentales, un programa en un lenguaje no procedimental es la "especificación del resultado deseado, en vez de la especificación de la acción requerida para conseguir el resultado" [COB85]. El software de soporte traduce la especificación del resultado en un programa máquina ejecutable. Hasta la fecha, los lenguajes de cuarta generación se han utilizado en aplicaciones de bases de datos y en otras áreas de procesamiento de datos para negocios.

Aplicaciones del software

El software puede aplicarse en cualquier situación en la que se haya definido previamente un conjunto específico de pasos procedimentales (es decir, un algoritmo). [Excepciones notables a esta regla son el software de los sistemas expertos y de redes neuronales]. Para determinar la naturaleza de una aplicación de software, hay dos factores importantes que se deben considerar : el *contenido* y el *determinismo de la información.*

El ***contenido*** se refiere al significado y a la forma de la información de entrada y de salida. Por ejemplo, muchas aplicaciones bancarias usan unos datos de entrada muy estructurados (una base de datos) y producen "informes" con determinados formatos. El software que controla una máquina automática (por ejemplo, un control numérico) actúa sobre elementos de datos discretos con una estructura limitada y produce órdenes concretas para la máquina en rápida sucesión.

El determinismo de la información se refiere a la predecibilidad del orden y del tiempo de llegada de los datos. Un programa de ingeniería acepta datos que están en un orden predefinido, ejecuta el algoritmo sin interrupción produce los datos resultantes en un informe o formato gráfico. Se dice que tales aplicaciones son determinadas. Un sistema operativo multiusuario, por otra parte, acepta entradas que tienen un contenido variado y que se producen en instantes arbitrarios, ejecuta algoritmos que pueden ser interrumpidos por condiciones externas y produce una salida que depende de una función del entorno y del tiempo. Las aplicaciones con estas características se dice que son indeterminadas.

Algunas veces es difícil establecer categorías genéricas para las aplicaciones del software que sean significativas. Conforme aumenta la complejidad del software, es más difícil establecer compartimentos nítidamente separados. Las siguientes áreas del software indican la amplitud de las posibilidades de aplicación.

Software de sistemas. El software de sistemas es un conjunto de programas que han sido escritos para servir a otros programas. Algunos programas de sistemas (p. ej.: compiladores, editores y utilidades de gestión de archivos) procesan estructuras de información complejas pero determinadas. Otras aplicaciones de sistemas (p. ej.: ciertos componentes del sistema operativo, utilidades de manejo de periféricos, procesadores de telecomunicaciones) procesan datos en gran medida indeterminados. En cualquier caso, el área del software de sistemas se caracteriza por una fuerte interacción con el hardware de la computadora; una gran utilización por múltiples usuarios; una operación concurrente que requiere una planificación, una compartición de recursos y una sofisticada gestión de procesos; unas estructuras de datos complejas y múltiples interfaces externas.

Software de tiempo real. El software que mide/analiza/controla sucesos del mundo real conforme ocurren, se denomina de *tiempo real.* Entre los elementos del software de tiempo real se incluyen: un componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, un componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo y un componente de monitorización que coordina todos los demás componentes, de forma que pueda mantenerse la respuesta en tiempo real (típicamente en el rango de 1 milisegundo a 1 minuto). Hay que tener en cuenta que el término "tiempo real" tiene un significado diferente de "interactivo" o "tiempo compartido". Un sistema de tiempo real debe responder dentro de unas ligaduras estrictas de tiempo. El tiempo de respuesta de un sistema interactivo (o de tiempo compartido) puede ser normalmente sobrepasado sin que se produzca ningún desastre.

Software de gestión. El procesamiento de información comercial constituye la mayor de las áreas de aplicación del software. Los "sistemas discretos" (por ejemplo: nóminas, cuentas de haberes/débitos, inventarios, etc.) han evolucionado hacia el software de sistemas de información de gestión (SIG), que accede a una o más bases de datos grandes que contienen información comercial. Las aplicaciones en este área reestructuran los datos existentes en orden a facilitar las operaciones comerciales o gestionar la toma de decisiones. Además de las tareas convencionales de procesamientos de datos, las aplicaciones de software de gestión también realizan cálculo interactivos (por ejemplo: el procesamiento de transacciones en puntos de ventas).

Software de ingeniería y científico. Caracterizado por los algoritmos de "manejo de números". Las aplicaciones van desde la astronomía a la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de las lanzaderas espaciales y desde la biología molecular a la fabricación automática.

Sin embargo, las nuevas aplicaciones del área de ingeniería/ciencia se han alejado de los algoritmos convencionales numéricos. El diseño asistido por computadora (del inglés, CAD), la simulación de sistemas y otras aplicaciones interactivas, han comenzado a tomar características del software de tiempo real e incluso del software de sistemas.

Software empotrado. Los productos inteligentes se han convertido en algo común en casi todos los mercados de consumo e industriales. El software empotrado reside en memoria de sólo lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo. El software empotrado puede ejecutar funciones muy limitadas y curiosas (por ejemplo: el control de las teclas de un horno de microondas) o suministrar una función significativa y con capacidad de control (por ejemplo: funciones digitales en un automóvil. tales como control de la gasolina, indicaciones en el salpicadero, sistemas de frenado, etc.).

Software de computadoras personales. El mercado del software de las computadoras personales ha germinado en la pasada década. El procesamiento de textos, las hojas de cálculo, los gráficos por computadora, entretenimientos, gestión de bases de datos, aplicaciones financieras, de negocios y personales, y redes o acceso a bases de datos externas son sólo algunas de los cientos de aplicaciones. De hecho, el software de las computadoras personales continúa representando uno de los diseños del software más innovadores en el campo del software.

Software de inteligencia artificial. El software de inteligencia artificial (IA) hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el calculo o el análisis directo. Actualmente, el área más activa de la IA es la de los *sistemas expertos* también llamados *sistemas basados en el conocimiento* [WAT85].

♦ SOFTWARE : UNA CRISIS EN EL HORIZONTE

Muchos observadores de la industria han caracterizado los problemas asociados con el desarrollo del software como una "crisis". Sin embargo, lo que realmente tenemos puede ser algo bastante diferente.

La palabra "crisis", se define en el *diccionario de Webster* como "un punto decisivo en el curso de algo ; momento, etapa o evento decisivo o crucial". Sin embargo, para el software no ha habido ningún "punto decisivo", ningún "momento decisivo", solamente un lento cambio evolutivo. En la industria del software hemos tenido una "crisis" que ha estado con nosotros cerca de 30 años y eso es una contradicción para el término.

Cualquiera que busque la palabra "crisis" en el diccionario encontrará otra definición "el punto decisivo en el curso de una enfermedad, cuando se ve más claro si el paciente vivirá o morirá". Esta definición puede darnos una pista sobre la verdadera naturaleza de los problemas que han acosado al desarrollo del software.

Nosotros ya hemos alcanzado la etapa de crisis en el software de computadoras. Lo que realmente tenemos es una *aflicción crónica* . La palabra "aflicción" se define en el Webster como "algo que causa pena o desastre". Pero la clave de nuestro argumento es la definición del adjetivo "crónica" : "muy duradero o que vuelve a aparecer con frecuencia ; continuando indefinidamente". Es bastante más preciso describir lo que hemos estado aguantando las tres últimas décadas como una aflicción crónica que como una crisis. No hay curas milagrosas, pero hay muchas formas con las que podemos reducir la pena a medida que nos esforcemos por descubrir un remedio.

Tanto si lo llamamos crisis del software como mal del software, el término alude a un conjunto de problemas que aparecen en el desarrollo del software de computadoras, Los problemas no se limitan al software que "no funciona correctamente".

Es más, el mal abarca los problemas asociados a cómo desarrollar software, cómo mantener el volumen cada vez mayor de software existente y cómo poder esperar mantenernos al corriente de la demanda creciente de software. Aunque se puede criticar la referencia a crisis o incluso a aflicción por ser melodramática, las frases resultan útiles por referirse a verdaderos problemas que se encuentran en todas las áreas del desarrollo del software.

Problemas

Los problemas que afligen al desarrollo del software se pueden caracterizar bajo muchas perspectivas diferentes, pero los responsables de los desarrollos de software se centran sobre los aspectos de "fondo" : (1) la planificación y estimación de costes son frecuentemente muy imprecisas; (2) la "productividad" de la comunidad del software no se corresponde con la demanda de sus servicios y (3) la calidad del software no llega a ser a veces ni aceptable. Se han experimentado desajustes en los costes de hasta un orden de magnitud. Se ha errado en la planificación en meses o años. Se ha hecho muy poco para mejorar la productividad de los trabajadores del software. Los errores en los nuevos programas producen en los clientes insatisfacción y falta de confianza.
Tales problemas son sólo las manifestaciones más visibles de otras dificultades del software :

- No tenemos tiempo de recoger datos sobre el proceso de desarrollo del software. Sin datos históricos como guía, la estimación no ha sido buena y los resultados previstos muy pobres. Sin una indicación sólida de la productividad, no podemos evaluar con precisión la eficacia de las nuevas herramientas, técnicas o estándares.

- La insatisfacción del cliente con el sistema "terminado" se produce demasiado frecuentemente. Los proyectos de desarrollo del software se acometen frecuentemente con sólo una vaga indicación de los requisitos del cliente. Normalmente, la comunicación entre el cliente y el que desarrolla el software es muy escasa.

- La calidad del software es normalmente cuestionable. Hemos empezado a comprender recientemente la importancia de la prueba sistemática y técnicamente completa del software. Están comenzando a emerger conceptos cuantitativos sólidos sobre la fiabilidad del software y las garantías de calidad [IAN84, EVA87].

- El software existente puede ser muy difícil de mantener. La tarea de mantenimiento del software se lleva la mayor parte de todo el dinero invertido en el software. El mantenimiento no se ha considerado un criterio importante en la aceptación del software.

Hemos presentado primero las malas noticias. Ahora las buenas : todos los problemas descritos anteriormente pueden corregirse. La clave está en dar un enfoque de ingeniería al desarrollo del software, junto con la mejora continua de las técnicas y de las herramientas.

Sin embargo, seguirá existiendo un problema (podríamos decir que son cosas de la vida). El software absorberá cada vez mayores porcentajes del coste de desarrollo global de los sistemas basados en computadora. En Estados Unidos se gastan cerca de 200 000 millones de dólares cada año en el desarrollo, compra y mantenimiento de software de computadora.

Causas

Los problemas asociados con la crisis del software se han producido por el propio carácter del software y por los errores de las personas encargados del desarrollo del mismo. Sin embargo, es posible que esperemos demasiado en demasiado poco tiempo.

Hemos tratado brevemente el carácter del software de computadora en la sección anterior. Recordemos : el software es un elemento lógico en vez de físico; por tanto, el éxito se mide por la calidad de una única entidad en vez de por muchas entidades fabricadas. El software no se rompe. Si se encuentran fallos, lo más probable es que se introdujeran inadvertidamente durante el desarrollo y no se detectaran durante la prueba. Reemplazamos las "partes defectuosas" durante el mantenimiento del software, pero tenemos muy pocas piezas de repuesto, incluso ninguna; es decir, el mantenimiento incluye normalmente la corrección o modificación del diseño.

La naturaleza lógica del software representa un desafío para la gente que lo desarrolla. Por primera vez hemos aceptado el reto de comunicarnos con un alienígena inteligente -una máquina. El desafío intelectual del desarrollo de software es seguramente una de las causas de la crisis del software, pero los problemas tratados anteriormente han sido causados por defectos humanos más mundanos.

Frecuentemente, los responsables del desarrollo del software han sido ejecutivos de nivel medio y alto, sin conocimientos de software. Un antiguo axioma de gestión dice : "Un buen gestor puede gestionar cualquier proyecto". Nosotros debemos añadir : "...Si está dispuesto a aprender las nuevas técnicas que pueden utilizarse para medir el desarrollo del proyecto, a aplicar métodos efectivos de control, a ignorar la mitología y a llegar a conocer una tecnología que cambia rápidamente". El gestor debe comunicarse con todos los componentes implicados en el desarrollo del software - clientes, realizadores del software,

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. Cuando esto ocurre, los problemas asociados con la crisis del software se multiplican.

Los trabajadores del software (en la pasada generación se llamaban programadores; en esta generación se ganará el título de ingenieros del software) han tenido muy poco entrenamiento formal en las nuevas técnicas de desarrollo de software. En muchas organizaciones reina una suave forma de anarquía. Cada individuo enfoca su tarea de "escribir programas" con la experiencia obtenida en trabajos anteriores. Algunas personas desarrollan un método ordenado y eficiente de desarrollo del software mediante prueba y error, pero muchos otros desarrollan malos hábitos que dan como resaltado una pobre calidad y mantenimiento del software.

Todos nos resistimos al cambio. Sin embargo, es verdaderamente irónico que, mientras el potencial de cálculo (hardware) experimenta enormes cambios, la gente del software, responsable de aprovechar dicho potencial, se oponga normalmente a los cambios cuando se discuten y se resista al cambio cuando se introduce. Puede que ésta sea la causa real de la crisis del software.

♦ MITOS DEL SOFTWARE

Muchas de las causas de la crisis del software se pueden encontrar en una mitología que surge durante los primeros años del desarrollo del software. A diferencia de los mitos antiguos, que ofrecían a los hombres lecciones dignas de tener en cuenta, los mitos del software propagaron información errónea y confusión. Los mitos del software tienen varios atributos que los hacen insidiosos; por ejemplo, aparecieron como declaraciones razonables de hechos (algunas

veces conteniendo elementos verdaderos); tuvieron un sentido intuitivo y frecuentemente fueron promulgados por expertos que "estaban al día".
Hoy, la mayoría de los profesionales competentes consideran a los mitos por lo que son - actitudes erróneas que han causado serios problemas, tanto a los gestores como a los técnicos. Sin embargo, las viejas actitudes y hábitos son difíciles de modificar y, cuando vamos hacia la quinta década del software, todavía se cree en algunos restos de los mitos del software.

Mitos de gestión. Los gestores con responsabilidad sobre el software, como los gestores en la mayoría de las disciplinas, están normalmente bajo la presión de cumplir los presupuestos, hacer que no se retrase el proyecto y mejorar la calidad. Igual que se agarra al vacío una persona que se ahoga, un gestor de software se agarra frecuentemente a un mito del software, aunque tal creencia sólo disminuya la presión temporalmente.

Mito. Tenemos ya un libro que está lleno de estándares y procedimientos para construir software. ¿No le proporciona ya a mi gente todo lo que necesita saber?
Realidad. Está muy bien que el libro exista, pero ¿se usa? ¿Conocen los trabajadores su existencia? ¿Refleja las prácticas modernas de desarrollo de software? ¿Es completo? En muchos casos, la respuesta a todas estas preguntas es "no".

Mito. Nuestra gente dispone de las herramientas de desarrollo de software más avanzadas ; después de todo les compramos las computadoras más nuevas.
Realidad. Se necesita mucho más que el último modelo de computadora grande (o de PC) para hacer desarrollo de software de gran calidad. Las herramientas de ingeniería del software asistida por computadora (CASE), aunque la mayoría todavía no se usen, son más importantes que el hardware para conseguir buena calidad y productividad.

Mito. Si fallamos en la planificación, podemos añadir más programadores y adelantar el tiempo perdido (el llamado algunas veces "concepto de la honda mongoliana".

Realidad. El desarrollo de software no es un proceso mecánico como la fabricación. En palabras de Brooks [BRO75] : "...añadir gente a un proyecto de software retrasado lo retrasa aún más". Al principio, esta declaración puede parecer un contrasentido. Sin embargo, cuando se añaden nuevas personas, la necesidad de aprender y comunicarse con el equipo puede y hace que se reduzca la cantidad de tiempo gastado en el desarrollo productivo. Puede añadirse gente, pero sólo de una manera planificada y bien coordinada.

Mitos del cliente. Un cliente que solicita una aplicación de software puede ser una persona del despacho de al lado, un grupo técnico de la sala de abajo, el departamento de ventas o una compañía exterior que solicita un software bajo contrato. En muchos casos, el cliente cree en los mitos que existen sobre el software, debido a que los gestores y trabajadores responsables hacen muy poco para corregir la mala información. Los mitos conducen a que el cliente se cree una falsa expectativa y, finalmente, quede insatisfecho con el que desarrolla el software.

Mito. Una declaración general de los objetivos es suficiente para comenzar a escribir los programas - podemos dar los detalles más adelante.

Realidad. Una mala definición inicial es la principal causa del trabajo baldío en software. Es esencial una descripción formal y detallada del ámbito de la información, funciones, rendimiento, interfaces, ligaduras del diseño y criterios de validación. Estas características pueden determinarse sólo después de una exhaustiva comunicación entre el cliente y el analista.

Mito. Los requisitos del proyecto cambian continuamente, pero los cambios pueden acomodarse fácilmente, ya que el software es flexible.

Realidad. Es verdad que los requisitos del software cambian, pero el impacto del cambio varía según el momento en que se introduzca.. Si se pone cuidado al dar la definición inicial, los cambios solicitados al principio pueden acomodarse fácilmente. El cliente puede revisar los requisitos y recomendar las modificaciones con relativamente poco impacto en el coste. Cuando los cambios se solicitan durante el diseño del software, el impacto en el coste crece rápidamente. Ya se han acordado los recursos a utilizar y se ha establecido un esqueleto del diseño. Los cambios pueden producir trastornos que requieran recursos adicionales e importantes modificaciones del diseño; es decir, coste adicional. Los cambios en la función, rendimiento, interfaces u otras características, durante la implementación (codificación y prueba) pueden tener un impacto importante sobre el coste. Cuando se solicitan al final de un proyecto, los cambios pueden producir un orden de magnitud mas caro que el mismo cambio pedido al principio.

Mitos de los desarrolladores. Los mitos en los que aún creen muchos desarrolladores se han ido fomentando durante cuatro décadas de cultura informática. Como ya indicamos anteriormente en este capítulo, durante los primeros días del desarrollo de software, la programación se veía como un arte. Las viejas formas y actitudes tardan en morir.

Mito. Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.

Realidad. Alguien dijo una vez: "Cuanto más pronto se comience a escribir código, más se tardará en terminarlo". Los datos industriales [LIE80] indican que entre el 50 y el 70 % de todo el esfuerzo dedicado a un programa se realizará después de que se le haya entregado al cliente por primera Vez.

Mito. Hasta que no tengo el programa "ejecutándose", realmente no tengo forma de comprobar su calidad.

Realidad. Desde el principio del proyecto se puede aplicar uno de los mecanismos más efectivos para garantizar la calidad del software - *la revisión técnica formal.* La revisión del software es un "filtro de calidad" que se ha comprobado que es más efectivo que la prueba, para encontrar ciertas clases de defectos en el software.

Mito. Lo único que se entrega al terminar el proyecto es el programa funcionando.

Realidad. Un programa que funciona es sólo una parte de una *configuración del software* que incluye todos los elementos que se muestran en la figura 1.4. La documentación es la base de un buen desarrollo y, lo que es más importante, proporciona guías para la tarea de mantenimiento del software.

Muchos profesionales del software reconocen la falacia de los mitos descritos anteriormente. Lamentablemente, las actitudes y métodos habituales fomentan una pobre gestión y malas prácticas técnicas, incluso cuando la realidad dicta un método mejor. El reconocimiento de las realidades del software es el primer paso hacia la formulación de soluciones prácticas para el desarrollo del mismo.

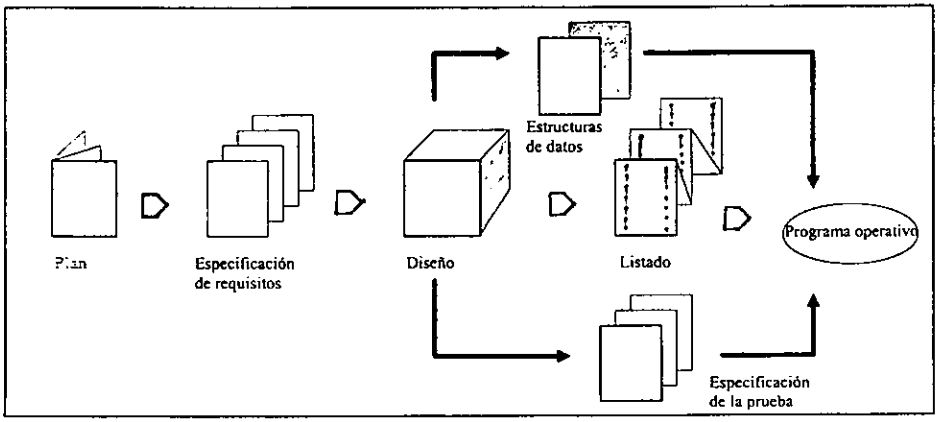


Figura 1.4 La configuración del software.

♦ PARADIGMAS DE LA INGENIERIA DEL SOFTWARE

El mal que ha infectado el desarrollo del software no va a desaparecer de la noche a la mañana. Reconocer los problemas y sus causas y demoler los mitos del software son los primeros pasos hacia las soluciones. Pero las propias soluciones tienen que proporcionar asistencia práctica a la persona que desarrolla software, mejorar la calidad del software y, por último, permitir al mundo del software mantenerse en paz con el mundo del hardware.

No existe un único entoque mejor para solucionar el mal del software. Sin embargo, mediante la combinación de métodos completos para todas las fases del desarrollo del software, mejores herramientas para automatizar estos métodos, bloques de construcción más potentes para la implementación del software, mejores técnicas para la garantía de calidad del software y una filosofía predominante para la coordinación, control y gestión, podemos conseguir una disciplina para el desarrollo del software - una disciplina llamada ***ingeniería del software***.

Ingeniería del software : una definición

Una de las primeras definiciones de ingeniería del software fue la propuesta por Fritz Bauer en la primera conferencia importante [NALJ69] dedicada al tema :

"El establecimiento y uso de principios de ingeniería robustos, orientados a obtener software económico que sea fiable y funcione de manera eficiente sobre máquinas reales.

Aunque se han propuesto muchas más definiciones generales, todas refuerzan la importancia de una disciplina de ingeniería para el desarrollo del software.

La ingeniería del software surge de la ingeniería de sistemas y de hardware. Abarca un conjunto de tres elementos clave - métodos, herramientas y procedimientos - que facilitan al gestor controlar el proceso del desarrollo del software y suministrar a los que practiquen dicha ingeniería las bases para construir software de alta calidad de una forma productiva. En los párrafos que siguen. examinaremos brevemente cada uno de estos elementos.

Los ***métodos*** de la ingeniería del 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 estructuras de datos, arquitectura de programas y procedimientos algorítmicos, codificación, prueba y mantenimiento. Los métodos de la ingeniería del software introducen frecuentemente una notación especial orientada a un lenguaje o gráfica y un conjunto de criterios para la calidad del software.

Las ***herramientas*** de la ingeniería del software suministran un soporte automático o semiautomático para los métodos. Hoy existen herramientas para soportar cada uno de los métodos mencionados anteriormente. Cuando se integran las herramientas de forma que la información creada por una herramienta pueda ser usada por otra, se establece un sistema para el soporte del desarrollo del software, llamado *ingeniería del software asistida por computadora* (del inglés, CASE). CASE combina software, hardware y bases de datos sobre ingeniería del software (una estructura de datos que contenga la información relevante sobre el análisis, diseño, codificación y prueba) para crear un entorno de ingeniería del software (por ejemplo, [BRE88] análogo al diseño/ingeniería asistido por computadora, CAD/CAE para el hardware.

Los *procedimientos* de la ingeniería del software son el pegamento que junta los métodos y las herramientas y facilita un desarrollo racional y oportuno del software de computadora. Los procedimientos definen la secuencia en la que se aplican los métodos, las entregas (documentos, informes, formas, etc.) que se requieren, los controles que ayudan a asegurar la calidad y coordinar los cambios, y las directrices que ayudan a los gestores del software a evaluar el progreso.

La ingeniería del software está compuesta por una serie de pasos que abarcan los métodos, las herramientas y los procedimientos antes mencionados. Estos pasos se denominan frecuentemente *paradigmas de la ingeniería del software.* La elección de un paradigma para la ingeniería del software se lleva a cabo de acuerdo con la naturaleza del proyecto y de la aplicación, los métodos y herramientas a usar y los controles y entregas requeridos. Tres son los paradigmas que se han tratado (y debatido) ampliamente; los dos más sobresalientes son los que se describen enseguida .

1) El ciclo de vida clásico

La Figura 1.5 ilustra el paradigma del ciclo de vida clásico para la ingeniería del software. Algunas veces llamado "modelo en cascada", el paradigma del ciclo de vida exige un enfoque sistemático y secuencial del desarrollo del software que comienza en el nivel del sistema y progresa a través del análisis, diseño, codificación, prueba y mantenimiento. Modelizado a partir del ciclo convencional de una ingeniería, el paradigma del ciclo de vida abarca las siguientes actividades:

* *Ingeniería y análisis del sistema.* Debido a que el software es siempre parte de un sistema mayor, el trabajo comienza estableciendo los requisitos de todos los elementos del sistema y luego asignando algún subconjunto de estos requisitos al software.

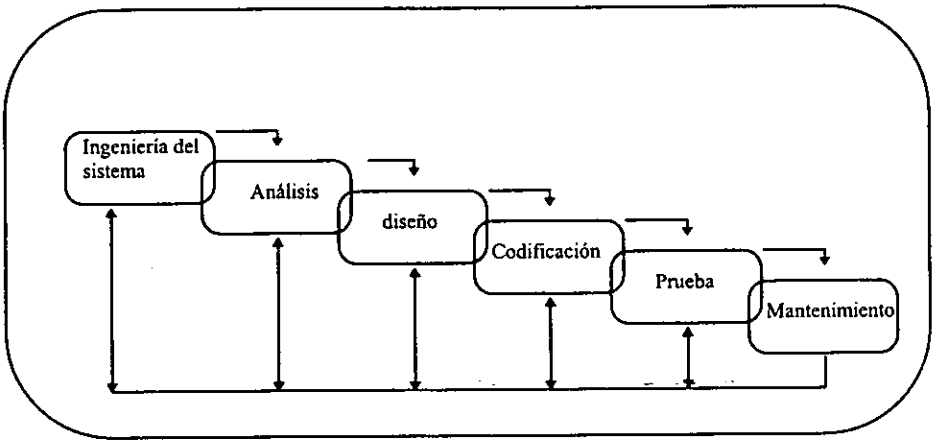


Figura 1.5 El ciclo de vida clásico.

Este planteamiento del sistema es esencial cuando el software debe interrelacionarse con otros elementos, tales como hardware, personas y bases de datos. La ingeniería y el análisis del sistema abarca los requisitos globales a nivel del sistema con una pequeña cantidad de análisisi y de diseño a un nivel superior.

* *Análisis de los requisitos del software.* El proceso de recopilación de los requisitos se contra e intensifica especialmente para el software. Para comprender la naturaleza de los programas que hay que construir, el ingeniero de software ("analista") debe comprender el ámbito de la información del software, así como la función, el rendimiento y, las interfaces requeridos. Los requisitos, tanto del sistema como del software, se documentan y se revisan con el cliente.

* *Diseño.* El diseño del software es realmente un proceso multipaso que se enfoca sobre cuatro atributos distintos del programa: la estructura de los datos, la arquitectura del software, el detalle procedimental y la caracterización de la interfaz. El proceso de diseño traduce los requisitos en una representación del software que pueda ser establecida de forma que obtenga la calidad requerida antes de que comience la codificación. Al igual que los requisitos, el diseño se documenta y forma parte de la configuración del software.

* *Codificación.* El diseño debe traducirse en una forma legible para la máquina. El paso de codificación realiza esta tarea. Si el diseño se realiza de una manera detallada, la codificación puede realizarse mecánicamente.

* *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 todas las sentencias se han probado, y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.

* *Mantenimiento.* El software, indudablemente, sufrirá cambios después de que se entregue al cliente (una posible excepción es el software empotrado). Los cambios ocurrirán debido a que se hayan encontrado errores, a que el software deba adaptarse a cambios del entorno externo (por ejemplo, un cambio solicitado debido a que se tiene un nuevo 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.

El ciclo de vida clásico es el paradigma más antiguo y más ampliamente usado en la ingeniería del software. Sin embargo, con el paso de unos cuantos años, se han producido críticas al paradigma, incluso por seguidores activos, que cuestionan su aplicabilidad a todas las situaciones. Entre los problemas que se presentan algunas veces, cuando se aplica el paradigma del ciclo de vida clásico, se encuentran :

a) Los proyectos reales raramente siguen el flujo secuencial que propone el modelo. Siempre hay iteraciones y se crean problemas en la aplicación del paradigma.

b) Normalmente, es difícil para el cliente establecer explícitamente al principio todos los requisitos. El ciclo de vida clásico lo requiere y tiene dificultades en acomodar posibles incertidumbres que pueden existir al comienzo de muchos proyectos.

c) El cliente debe tener paciencia. Hasta llegar a las etapas finales del desarrollo del proyecto, no estará disponible una versión operativa del programa. Un error importante no detectado hasta que el programa esté funcionando puede ser desastroso.

Cada uno de estos problemas es real. Sin embargo, el paradigma clásico del ciclo de vida tiene un lugar definido e importante dentro del trabajo realizado en ingeniería del software. Suministra una plantilla en la que pueden colocarse los métodos para el análisis, diseño, codificación prueba y mantenimiento. Además, veremos que los pasos del paradigma clásico del ciclo de vida son muy similares a los pasos genéricos (sección 1.6) aplicables a todos los paradigmas de ingeniería del software. El ciclo de vida clásico sigue siendo el modelo procedimental más ampliamente usado por los ingenieros del software. A pesar de sus inconvenientes, es significativamente mejor que desarrollar el software sin guías.

2) Construcción de prototipos

Normalmente un cliente define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el programador puede no estar seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma en que debe realizarse la interacción hombre-máquina. En estas y muchas otras situaciones, puede ser mejor método de ingeniería del software la construcción de un prototipo.

La construcción de prototipos es un proceso que facilita al programador la creación de un modelo del software a construir. El modelo tomará una de las tres formas siguientes : (1) un prototipo en papel o un modelo basado en PC que describa la interacción hombre-máquina, de forma que facilite al usuario la comprensión de cómo se producirá tal interacción; (2) un prototipo que implemente algunos subconjuntos de la función requerido del programa deseado, o (3) 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 figura 1.6 muestra la secuencia de sucesos del paradigma de construcción de prototipos. Como en todos los métodos de desarrollo de software, la construcción de prototipos comienza con la recolección de los requisitos. El técnico y el cliente se reunen y definen los objetivos globales para el software, identifican todos los requisitos conocidos y perfilan las áreas en donde será necesario una mayor definición. 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 del software. Si se va a construir un prototipo que funcione, el realizador intenta hacer uso de fragmentos de programas existentes o aplica herramientas (por ejemplo, generadores de informes, gestores de ventanas, etc.) que faciliten la rápida generación de programas que funcionen.

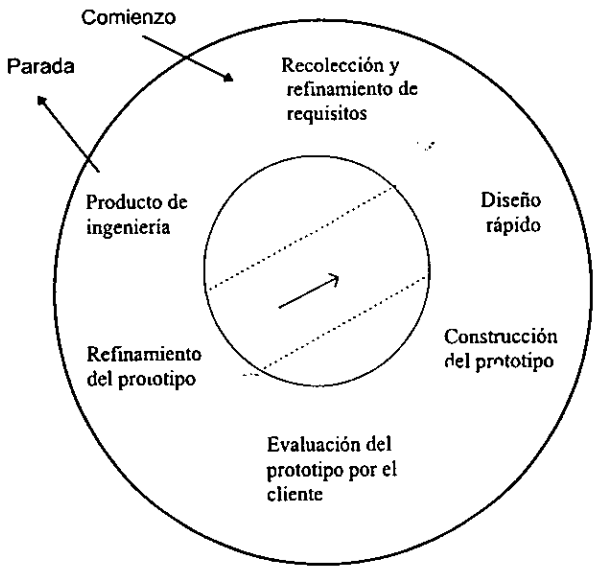


Figura 1.6. Creación de prototipos.

Pero, ¿qué debemos hacer con el prototipo cuando ya ha servido para el propósito establecido? Brooks [BRO75] nos da una respuesta :

"En la mayoría de los proyectos, el primer sistema construido apenas es utilizable. Puede ser demasiado lento, demasiado grande, difícil de usar o las tres cosas. No hay más alternativa que comenzar de nuevo y construir una versión rediseñada que resuelva los problemas que se presenten... Cuando se utiliza un nuevo concepto de sistema o de tecnología, hay que construir un sistema para desecharlo, porque incluso la mejor planificación no puede asegurar que vaya a ser bueno la primera vez. Por tanto, la cuestión no es si hay que construir un sistema piloto y tirarlo. Se tirará. La única cuestión es si planificar de antemano la construcción de algo que se va a desechar, o prometer entregar el desecho a los clientes..."

▸ CAPITULO II : CALIDAD DEL SOFTWARE

Todos los métodos, herramientas y procedimientos descritos en esta tesis van orientados a un único fin: producir software de gran calidad. Todavía muchos lectores estarán preocupados por la pregunta: "¿Qué es software de calidad?". Philip Crosby[6] en su famoso libro sobre calidad, expone esta situación:

"El problema de la gestión de la calidad no es lo que la gente no sabe sobre ella. El problema es lo que creen que saben...
Salvando las diferencias, la calidad tiene mucho en común con el sexo. Todo el mundo lo quiere. (Bajo ciertas condiciones, por supuesto). Todo el mundo cree que lo conoce. (Incluso aunque no quiera explicarlo). Todo el mundo piensa que su ejecución sólo es cuestión de seguir las inclinaciones naturales. (Después de todo, nos las arreglamos de alguna forma). Y, por supuesto, la mayoría de la gente piensa que, los problemas en estas áreas están producidos por otra gente. (Como si sólo ellos se tomaran el tiempo para hacer las cosas bien)".

La situación de este capítulo puede hacer creer al lector que la garantía de la calidad del software es algo en lo que se empieza a pensar una vez que se ha generado el código. Nada más lejos de la verdad. La garantía de calidad del software (SQA)[1] es una "actividad de protección" que se aplica a lo largo de todo el proceso de ingeniería software. La SQA engloba: (1) métodos y herramientas de análisis, diseño, codificación y prueba; (2) revisiones técnicas formales que se aplican durante cada paso de la ingeniería del software; (3) una estrategia de prueba multiescalada; (4) el control de la documentación del software y de los cambios realizados; (5) un procedimiento que asegure un ajuste a los estándares de desarrollo del software (cuando sea posible); y (6) mecanismos de medida y de información.

---

[6] Crosby, P. Quality is Free, Mcgraw-Hill, 1979.
[1] N.T. : Utilizaremos las siglas inglesas de ("Software Quality Assurance") por estar bastante extendidas en el campo de la ingeniería del software.

En este capítulo examinaremos el significado del evasivo término calidad del software y discutiremos los procedimientos y las medidas que ayudan a garantizar que la calidad sea un resultado natural de la ingeniería del software.

♦ CALIDAD DEL SOFTWARE Y GARANTIA DE CALIDAD DEL SOFTWARE

Hasta el realizador de software más experimentado estará de acuerdo con que el software de alta calidad es una meta importante. Pero, ¿como definimos la calidad? Un bromista dijo una vez: "Cualquier programa hace algo bien, lo que puede pasar es que no sea lo que nosotros queremos que haga".

En los libros se han propuesto muchas definiciones de la calidad del software. Por lo que aquí respecta, la calidad del software se definirá como:

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

No hay duda de que la anterior definición puede ser modificada o ampliada. De hecho, no tendría fin una discusión sobre una definición definitiva de la calidad del software. Para los Propósitos de esta tesis, la anterior definición sirve para hacer hincapié en cuatro puntos importantes:

1. El software debe satisfacer al cliente en todos sus aspectos.

2. Los requisitos del software son la base de las medidas de la calidad. La falta de concordancia con los requisitos es una falta de calidad.

3. Los estándares especificados definen un conjunto de criterios de desarrollo que guían la forma en que se aplica la ingeniería del software. Si no se siguen esos criterios, casi siempre habrá falta de calidad.

4. Existe un conjunto de requisitos implícitos que a menudo no se mencionan (p. ej.: 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 queda en entredicho.

La calidad del software es una compleja mezcla de ciertos factores que varían para las diferentes aplicaciones y los clientes que las solicitan, este último por su parte , resulta un factor de suma importancia a lo largo del proceso de desarrollo de la aplicación, ya que de su involucramiento dependerá la calidad que el espera recibir en el producto final.

En las siguientes secciones, se identifican los factores de la calidad del software y se describen las actividades humanas requeridas para alcanzarlos.

FACTORES QUE DETERMINAN LA CALIDAD DEL SOFTWARE

Los factores que afectan a la calidad del software se pueden clasificar en dos grandes grupos: (1) factores que pueden ser medidos directamente (p. ej.: errores/KLDC/unidad de tiempo) y (2) factores que sólo pueden ser medidos indirectamente (p. ej.: facilidad de uso o de mantenimiento). En cualquiera de los dos casos se puede medir. Debemos comparar el software (documentos, programas, etc.) con alguna referencia y llegar a una indicación de la calidad

McCall y sus colegas [MCC77] han propuesto una útil clasificación de los factores que afectan a la calidad del software. Estos factores de calidad del software que aparecen en la Figura 2.1, se centran en tres aspectos importantes de un producto de software: sus características operativas, su capacidad de soportar los cambios y su adaptabilidad a nuevos entornos.

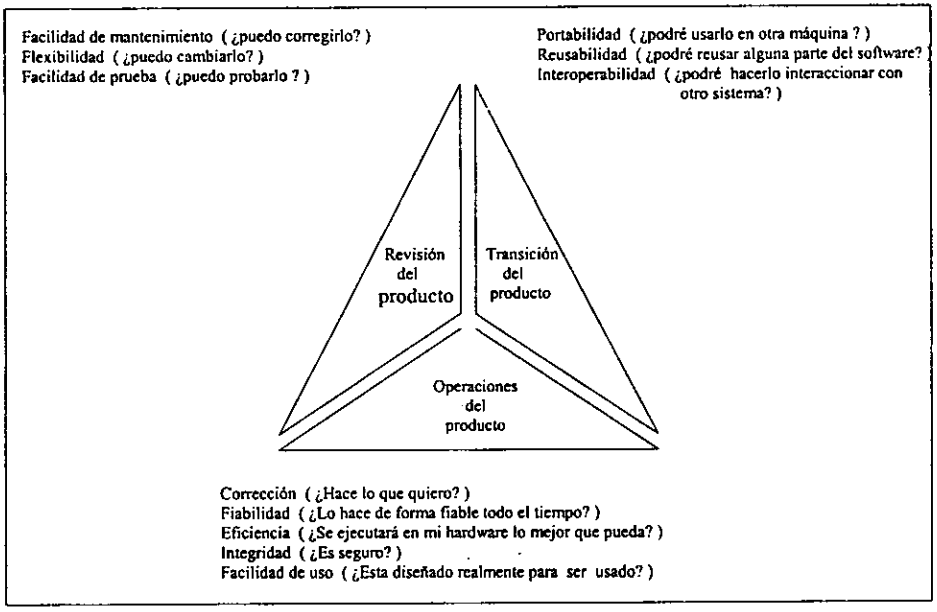


Figura 2.1 Factores de calidad del software de McCall.

Para los factores indicados en la figura se proporcionan las siguientes descripciones:

Corrección.- El grado en que un programa satisface sus especificaciones y consigue los objetivos de la misión encomendada por el cliente.

Fiabilidad.- El grado en que se puede esperar que un programa lleve a cabo sus funciones esperadas con la precisión requerida.

Eficiencia.- La cantidad de recursos de computadora y de código requeridos por un programa para llevar a cabo sus funciones.

Integridad.- El grado en que puede controlarse el acceso al software o a los datos, por personal no autorizado.

Facilidad de uso.- El esfuerzo requerido para entrar a un programa, trabajar con él, preparar su entrada e interpretar su salida.

Facilidad de mantenimiento.- El esfuerzo requerido para localizar y arreglar un error en un programa.

Flexibilidad.- El esfuerzo requerido para modificar un programa operativo.

Facilidad de prueba.- El esfuerzo requerido para probar un programa de forma que asegure que realiza su función requerida

Portabilidad.- El esfuerzo requerido para transferir el programa desde un hardware y/o un entorno de sistemas de software a otro.

Reusabilidad.- El grado en que un programa (o partes de un programa) se puede reusar en otras aplicaciones. Esto va relacionado con El empaquetamiento y el alcance de las funciones que realiza el programa.

Facilidad de interoperación.- El esfuerzo requerido para acoplar un sistema a otro.

Se define además un conjunto de métricas usadas para desarrollar expresiones para cada uno de los factores de acuerdo con la siguiente relación :

$$F_c = c_1 \times m_1 + c_2 \times m_2 + \ldots + c_i \times m_i$$

donde F_c es un factor de calidad del software, c_i son coeficientes de regresión y m_i son la métricas que afectan al factor de calidad, cuya escala de graduación va de 0 a 10. con las siguientes métricas:

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 el que se usan el ancho de banda, los protocolos y las interfaces estándar.

Completitud.- El grado en el 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 el 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 los componentes del programa.

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

Seguridad.- La disponibilidad de los 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 seguir la pista a 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 perrnitir que nuevos usuarios apliquen el sistema.

La relación entre los factores de calidad del software y las métricas que se acaban de indicar se muestra en la tabla de la Figura 2.2. Se debe recordar que el peso dado a cada métrica dependerá del producto en particular.

Métrica de calidad del software / Factor de calidad	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										
Formación											X

Figura 2.2 Relación entre las métricas y los factores de calidad del software

Los factores de calidad descritos por McCall y sus colegas [MCC77] representan una de tantas "listas de comprobación" que se han sugerido para la calidad del software. Hewlett-Packard [GRA87] ha desarrollado un conjunto de factores de calidad del software cuyas siglas son *FURPS* -por funcionalidad, facilidad de uso, fiabilidad, rendimiento y capacidad de soporte (en inglés). Los factores de calidad FURPS se han obtenido libremente de trabajos anteriores y se definen los siguientes atributos para cada uno de los cinco factores principales:

* **La funcionalidad** se obtiene mediante la evaluación del conjunto de características y de posibilidades del programa, la generalidad de las funciones que se entregan y la seguridad de todo el sistema.
* **La facilidad** *de* uso se calcula considerando los factores humanos, la estética global, la consistencia y la documentación.
* **La fiabilidad** se calcula midiendo la frecuencia de fallos y su importancia, la eficacia de los resultados de salida, el tiempo medio entre fallos (TMEF), la posibilidad de recuperarse a los fallos y la previsibilidad del programa.
* **El rendimiento** se mide mediante la evaluación de la velocidad de proceso, el tiempo de respuesta, el consumo de recursos, el rendimiento total de procesamiento y la eficiencia.
* **La capacidad de soporte** combina la posibilidad de ampliar el programa (extensibilidad), la adaptabilidad y la utilidad (estos tres atributos representan un término más común facilidad de mantenimiento), además de la facilidad de prueba, la compatibilidad, la posibilidad de configuración, la facilidad con la que se puede instalar un sistema y la facilidad con la que se pueden localizar los problemas.

GARANTÍA DE CALIDAD DEL SOFTWARE

La garantía de calidad es una actividad esencial en cualquier empresa que produce productos que van a ser usados por otros. Antes del siglo veinte, la garantía de calidad era responsabilidad única de la persona que construía el producto. La primera función de control y de garantía de calidad formal fue introducida por los laboratorios Bell en 1916 y se extendió rápidamente por todo el mundo de las manufacturas. Hoy en día, cada empresa tiene un mecanismo que asegura la calidad de sus productos. De hecho, durante la pasada década se ha usado ampliamente, como táctica de mercado, la declaración explícita de mensajes que ponían de manifiesto la calidad ofrecida por las empresas.
La historia de la garantía de calidad en el desarrollo de software ha ido paralela a la historia de la calidad en la fabricación de hardware. Durante los primeros años de la informática (los años 50 y 60), la calidad era responsabilidad únicamente del programador. Durante los años 70 se introdujeron estándares de garantía de calidad para el software en los contratos militares de desarrollo de software y se han extendido rápidamente en los desarrollos de software del mundo comercial [IEEE89].

La garantía de calidad del software (SQA) es un "planificado y sistemático diseño de acciones" [SCII87] que se requieren para asegurar la calidad del software. El alcance de la responsabilidad de la garantía de calidad se puede caracterizar de la mejor forma parafraseando un popular anuncio de automóviles: "La calidad es el trabajo Nº 1 . Lo que esto implica en el desarrollo de software es qué la responsabilidad de la garantía de calidad del software corresponde a muchos constituyentes de una organización -ingenieros de software, gestores del proyecto, clientes, comerciales y personas que trabajan dentro del grupo de SQA.

El grupo de SQA sirve como representación del cliente dentro de la casa. Es decir, la gente que lleva a cabo la SQA debe mirar el software desde el punto de

vista del cliente. ¿Satisface de forma adecuada el software los factores de calidad?, ¿Se ha realizado el desarrollo del software de acuerdo con estándares preestablecidos?, ¿Han desempeñado apropiadamente sus papeles las disciplinas técnicas como parte de la actividad de SQA? El grupo de SQA intenta responder a éstas y otras cuestiones para asegurar que se mantiene la calidad del software.

Actividades de SQA

Dentro de las actividades de aseguramiento de la calidad de software se tienen :

1) aplicación de métodos técnicos

2) realización de revisiones técnicas formales.

3) prueba del software.

4) ajuste a los estándares.

5) control de cambios.

6) mediciones.

7) registro y realización de informes.

La calidad del software debe estar diseñada para el producto o sistema, por esta razón el aseguramiento de la calidad comienza realmente con un conjunto de herramientas y métodos técnicos que ayudan al analista a conseguir una especificación de alta calidad y un diseño de alta calidad.

La adopción de un modelo apropiado de ciclo de vida para el desarrollo de software es un paso importante hacia el mejoramiento de la calidad. Es el primer nivel para tener bajo control el proceso de desarrollo de software.

Una vez que se ha creado una especificación (o prototipo) y un diseño, debe ser asegurada su calidad. La actividad central que permite garantizar la calidad es la revisión técnica formal (RTF), la cual es una especie de reunión del personal técnico con el único propósito de descubrir problemas de calidad. En muchas situaciones se ha visto que las revisiones son tan efectivas como la prueba para descubrir los defectos en el software.

La prueba del software combina una estrategia de múltiples pasos con una serie de métodos de casos de prueba que ayudan a asegurar una efectiva detección de errores.
El grado de aplicación de procedimientos y estándares en el proceso de la ingeniería del software varía de empresa a empresa. En muchos casos, los estándares vienen dados por los clientes o por mandamientos de regulación. En otras situaciones, los estándares se imponen por sí solos. Para los estándares formales (escritos), se debe establecer una actividad de aseguramiento de la calidad para garantizar que estos se siguen. La verificación del seguimiento de estándares puede ser llevada a cabo por los encargados del desarrollo de software como parte de una revisión técnica formal o, en situaciones en que se requiera una verificación del seguimiento independiente por el grupo de aseguramiento de la calidad (SQA), mediante su propia auditoría.

Una de las principales amenazas para la calidad se deriva de los aparentemente benignos cambios. Cada cambio realizado sobre el software en potencia puede introducir errores o crear efectos laterales que propaguen errores. El formalizar el control de cambios mediante las peticiones formales de cambio, evaluar la naturaleza del cambio y controlar el impacto del cambio contribuyen directamente a la calidad del software.

La medición es una actividad integral para cualquier disciplina. Un objetivo importante para el aseguramiento de la calidad, es seguir la pista a la calidad del software y evaluar el impacto de los cambios de metodología y de procedimiento que intentan mejorar la calidad del software.
El registro de la información y la generación de informes para el aseguramiento de la calidad del software dan procedimientos para la recolección y divulgación de información acerca de la calidad. Los resultados de las revisiones, auditorías, control de cambios, prueba y otras actividades de aseguramiento de la calidad deben convertirse en una parte del registro histórico de un proyecto y deben ser divulgados a la plantilla de desarrollo para que tengan conocimiento de ellos. Por ejemplo, los resultados de cada RTF de un diseño procedimental se registran y se guardan en una "carpeta" que contenga toda la información técnica y de SQA sobre cada módulo.

- **REVISIONES DEL SOFTWARE**

Las revisiones del software son un "filtro" para el proceso de ingeniería del software. Las revisiones se aplican en varios momentos del desarrollo del software y sirven para detectar defectos que puedan así ser eliminados. Las revisiones del software sirven para "purificar" las actividades de ingeniería del software que hemos denominado análisis, diseño y codificación. Freedman y Weinberg argumentan de la siguiente forma la necesidad de revisiones :

El trabajo técnico necesita ser revisado por la misma razón que los lápices necesitan gomas: errar es humano. La segunda razón por la que necesitamos revisiones técnicas es que, aunque la gente es buena cazando algunos de sus propios errores, algunas clases de errores se le pasan por alto más fácilmente al que los origina que a otras personas

Una revisión, cualquier revisión, es una forma de aprovechar la diversidad de un grupo de personas para :

1. Señalar la necesidad de mejoras en el producto de una sola persona o un equipo
2. Confirmar las partes de un producto en las que no es necesaria o no es deseable una mejora.
3. Conseguir un trabajo técnico de una calidad más uniforme, o al menos más predecible, que la que puede ser conseguida sin revisiones, con el fin de hacer más manejable el trabajo técnico.

Existen muchos tipos diferentes de revisiones que se pueden llevar adelante como parte de la ingeniería del software. Cada una tiene su lugar. Una reunión informal alrededor de la máquina de café es una forma de revisión, si se discuten problemas técnicos. Una presentación formal de un diseño de software a una audiencia de clientes, ejecutivos y personal técnico es una forma de revisión. Sin embargo, en esta tesis nos centraremos en la revisión técnica formal (RTF) a veces denominada inspección. Una revisión, técnica formal es el filtro más efectivo desde el punto de vista de garantía de calidad. Llevada a cabo por ingenieros de software para los ingenieros de software, la RFT es un medio efectivo para mejorar la calidad del software.

Impacto de los defectos del software sobre el costo

El beneficio más obvio de las revisiones técnicas formales es el pronto descubrimiento de los defectos del software, de forma que cada defecto pueda ser corregido antes de llegar al siguiente paso del proceso de ingeniería del software. Por ejemplo. varios estudios industriales (TRW, Nippon Electric, Mitre Corp., entre otros) indican que las actividades de diseño introducen entre el 50 y el 65 por 100 de todos los errores (defectos) de la fase de desarrollo del proceso de ingeniería del software. Sin embargo, las técnicas de revisión formal se han mostrado efectivas, hasta en un 75 por 100 de los casos, para el descubrimiento de

flaquezas del diseño. Al detectar y eliminar un gran porcentaje de esos errores, el proceso de revisión, reduce substancialmente el costo de los siguientes pasos de las fases de desarrollo y mantenimiento.

Amplificación y eliminación de defectos.

Se puede usar un modelo de amplificación de defectos para ilustrar la generación y detección de errores durante los pasos de diseño preliminar, diseño detallado y codificación del proceso de ingeniería del software. Durante cada paso se pueden generar errores inadvertidamente. La revisión puede fallar en descubrir nuevos errores y errores de pasos anteriores, resultando en un mayor número de errores que pasan inadvertidos. En algunos casos, los errores que pasan inadvertidos desde pasos anteriores se amplifican (factor de amplificación x) con el trabajo actual. Las subdivisiones de los cuadros representan cada una de esas características y el porcentaje de eficiencia en la detección de errores, una función de la profundidad de la revisión.

Para llevar a cabo revisiones, el equipo de desarrollo debe dedicar tiempo, esfuerzo y dinero. Sin embargo, los resultados del ejemplo anterior no dejan duda de que hemos encontrado el síndrome de "pague ahora o pague, después, mucho más". Las revisiones técnicas formales (para el diseño y otras actividades técnicas) producen un beneficio en costo demostrable. Deben llevarse a cabo.

- **Revisiones técnicas formales**

Una revisión técnica formal (RTF) es una actividad de garantía de calidad del software que es llevada a cabo por los profesionales de la ingeniería del software. Los objetivos de la RTF son : descubrir errores en la función, la lógica o la implementación de cualquier representación del software así como verificar que el algoritmo sea claro y sin ambigüedades.

Directrices para la revisión

Se deben establecer de antemano directrices para conducir las revisiones técnicas formales, distribuyéndolas después entre los revisores. A menudo, una revisión incontrolada puede ser peor que no hacer ningún tipo de revisión.

A continuación se muestra un conjunto mínimo de directrices para las revisiones técnicas formales :

* *Revisar el producto, no al productor.*

Una RTF involucra gente y egos. Llevada a cabo adecuadamente, la RTF debe llevar a todos los participantes a un sentimiento agradable de estar cumpliendo su deber. Si se lleva a cabo incorrectamente, la RTF puede tomar el aura de inquisición. Se deben señalar los errores educadamente ; el tono de la reunión debe ser distendido y constructivo no debe intentarse dificultar a batallar. El jefe de revisión debe moderar la reunión para garantizar que se mantiene un tono y una actitud adecuados y debe inmediatamente cortar cualquier revisión que haya escapado al control.

* *Fijar la agenda y mantenerla*

Un mal de las reuniones de todo tipo es la deriva. La RTF debe seguir un plan de trabajo concreto. El jefe de revisión es el que carga con la responsabilidad de mantener el plan de la reunión y no debe tener miedo de cortar a la gente cuando se empiece divagar.

* *Limitar el debate y las impugnaciones*

Cuando un revisor ponga de manifiesto una pega, podrá no haber unanimidad sobre su impacto. En lugar de perder el tiempo debatiendo la cuestión, debe registrarse el hecho y dejar que la discusión se lleve a cabo en otro momento.

* *Enunciar áreas de problemas, pero no intentar resolver cualquier problema que se ponga de manifiesto.*

Una revisión no es una sesión de resolución de problemas. A menudo, la resolución de los problemas puede ser encargada al productor por sí solo o con la ayuda de la persona. La resolución de los problemas debe ser pospuesta para después de la reunión de revisión.

* *Tomar notas escritas*

A veces es buena idea que el registrador tome las notas en una pizarra, de forma que las declaraciones o la asignación de prioridades pueda ser comprobada por el resto de los revisores, a medida que se va registrando la información.

* *Limitar el número de participantes e insistir en la preparación anticipada.*

Dos ojos ven más que uno, pero 14 no son más necesarios que cuatro. Se ha de mantener el número de participantes en el mínimo necesario. Además, todos los miembros del equipo de revisión deben prepararse por anticipado. El jefe de revisión debe solicitar comentarios escritos (que muestren que cada revisor ha revisado el material).

* *Desarrollar una lista de comprobaciones para cada producto que haya de ser revisado.*

Una lista de comprobaciones ayuda al jefe de revisión a estructurar la reunión de RTF, y ayuda a cada revisor a centrarse en los asuntos importantes. Se deben desarrollar listas de comprobaciones para los documentos de análisis, de diseño, de codificación e incluso de prueba.

* *Disponer recursos y una agenda para las RTF.*

Para que las RTF sean efectivas, se deben planificar como una tarea del proceso de Ingeniería del software. Además, se debe trazar un plan de actuación para las modificaciones inevitables que aparecen como resultado de una RTF.

* *llevar a cabo un buen entrenamiento de todos los revisores.*

Por razones de efectividad, todos los participantes en la revisión deben recibir algún entrenamiento formal. El entrenamiento se debe basar en los aspectos relacionados con el proceso, así como en las consideraciones de psicología humana que atañen a la revisión. Freedman y Weinberg estiman en un mes la curva de aprendizaje para cada 20 personas que vayan a participar de forma efectiva en las revisiones

* *Repasar las revisiones anteriores*

Las sesiones de información pueden ser beneficiosas para descubrir problemas en el propio proceso le revisión. El primer producto que se haya revisado puede establecer las propia directivas de revisión.

Una lista de comprobaciones para la revisión

Se pueden realizar revisiones técnicas formales durante cada paso del proceso de ingeniería del Software. En esta sección presentamos una breve lista de comprobaciones que se deben usar para garantizar los productos que se despachan como parte de un desarrollo de software Las listas de comprobaciones no pretenden ser completas sino más bien proporcionar un punto de partida para cada revisión.

Ingeniería del sistema. *La especificación del sistema* asigna la función y el rendimiento de muchos elementos del sistema. Por tanto, la revisión del sistema involucra muchos componentes que se centran cada uno en su propia área que le concierne. Los grupos de ingeniería del software y de ingeniería del hardware se centran en las aspiraciones del software y del hardware, respectivamente La garantía de calidad evalúa los requisitos de validación a nivel del sistema y el servicio de campo examina los requisitos para llevar a cabo diagnósticos.

Una vez realizadas todas las revisiones, se lleva a cabo una reunión de revisión más amplia con representantes de cada componente, con el fin de asegurar una buena comunicación de lo que a cada uno le concierne. La siguiente lista de comprobaciones cubre algunas de las áreas concernientes más importantes :

1. ¿Se han definido funciones principales de forma delimitada y sin ambigüedad?

2. ¿Se han definido interfaces entre los elementos del sistema?

3. ¿Se han establecido límites de prestaciones para el sistema como un todo y para cada elemento?

4. ¿Se han establecido restricciones en el diseño de cada elemento ?

5. ¿Se ha elegido la mejor alternativa?

6. ¿Es la solución técnicamente factible?

7. ¿Se ha establecido un mecanismo de validación y verificación?

8. ¿Existe consistencia entre todos los elementos del sistema?

Planificación del proyecto de software. Las estimaciones de recursos, costos y tiempos para el desarrollo, llevadas a cabo en la planificación del proyecto de software, se basan en la asignación del software establecida dentro de la actividad de la ingeniería del sistema. La revisión del plan del proyecto de software debe intentar establecer el grado de riesgo. Se puede seguir la siguiente Lista de comprobaciones:

1. ¿Se ha definido el alcance del software de forma limitada y sin ambigüedades?

2. ¿Es clara la terminología?

3. ¿Son adecuados los recursos para ese alcance?

4. ¿Están fácilmente disponibles los recursos?

5. ¿Se han definido los riesgos en todas las categorías importantes?

6. ¿Existe un plan de gestión de riesgos?

7. ¿Se han definido las tareas y su secuencia adecuadamente? ¿Es razonable el paralelismo en función de los recursos disponibles?

8. ¿Es razonable la base de la estimación de costos? ¿Se han utilizado dos métodos independientes para la estimación de costos?

9. ¿Se han utilizado datos históricos de productividad y de calidad

10. ¿Se han reconciliado las diferencias entre estimaciones?

11. ¿Son realistas el presupuesto y la fecha tope preestablecidos?

12. ¿Es consistente la agenda?

Análisis de requisitos del software. Las revisiones del análisis de requisitos del software se centran en el seguimiento de los requisitos del sistema y de la consistencia y corrección de la representación.

Para los requisitos de un gran sistema se llevan a cabo numerosas RTFs, pudiendo verse ampliadas por las revisiones y evaluaciones de prototipos, así como por las reuniones con los clientes. Durante las RTFs del análisis se consideran los siguientes aspectos:

1. ¿Es completo, consistente y exacto el análisis del campo de información?

2. ¿Es completa la partición del problema?

3. ¿Están definidas adecuadamente las interfaces internas y externas?

4. ¿Refleja el modelo de datos correctamente los datos, sus atributos y sus relaciones?

5. ¿Se pueden seguir todos los requisitos a nivel del sistema?

6. ¿Se ha realizado un prototipo para el usuario?

7. ¿Son alcanzables las prestaciones con las restricciones impuestas por otros elementos del sistema?

8. ¿Son consistentes los requisitos con la planificación y los recursos?

9. ¿Son completos los criterios de validación?

Diseño del software. Las revisiones del diseño del software se centran en el diseño de datos, el diseño arquitectónico y el diseño procedimental. En general, se pueden realizar dos tipos de revisiones del diseño. La revisión del diseño preliminar confirma la traducción de los requisitos al diseño y se centra en la arquitectura del software. La segunda revisión, a menudo denominada inspección del diseño, centra su atención en la corrección procedimental de los algoritmos, tal y como están implementados en los módulos del programa. Para estas revisiones son útiles las siguientes listas de comprobaciones :

Para la revisión del diseño preliminar :

1. ¿Están reflejados los requisitos del software en la arquitectura del software?
2. ¿Se ha conseguido una modularidad efectiva? ¿Son funcionalmente independientes los módulos.
3. ¿Depende de algunos factores la arquitectura del programa?
4. ¿Se han definido las interfaces para los módulos y los elementos externos del sistema?
5. ¿Es consistente la estructura de datos con el ámbito de información?
6. ¿Es consistente la estructura de datos con los requisitos del software ?
7. ¿Se ha considerado la facilidad de mantenimiento?
8. ¿Se han evaluado explícitamente los factores de calidad ?

para la inspección del diseño :

1. ¿Realiza el algoritmo la función deseada?
2. ¿ Es el algoritmo lógicamente correcto?
3. ¿Es consistente la interfaz con el diseño Arquitectónico?
4. ¿Es razonable la complejidad lógica?
5. ¿Se ha especificado el tratamiento de errores y la "tolerancia a errores"?
6. ¿Se han definido adecuadamente las estructuras de datos locales?
7. ¿Se usa lógica compuesta o inversa?

8. ¿Se han utilizado ampliamente las construcciones de la programación estructurada?
9. ¿ Es adecuado el nivel de detalle del diseño para el lenguaje de implementación?
10. ¿Se han utilizado características dependientes del sistema operativo o del lenguaje?
11. ¿Se ha tenido en cuenta la facilidad de mantenimiento?

Codifcación. Aunque la codificación es un resultado mecánico de diseño procedimental, se pueden introducir errores al traducir el diseño a un lenguaje de programación. Eso es particularmente cierto si el lenguaje de programación no soporta directamente las estructuras de datos y de control representadas en el diseño. Un recorrido por el código puede ser un medio efectivo para descubrir estos errores de traducción.

La lista de comprobaciones que viene a continuación asume que se ha llevado a cabo una inspección del código y que se ha establecido la validez algorítmica como parte de la RTF del diseño.

1. ¿Se ha traducido adecuadamente el diseño al código? (Durante esta revisión deben estar disponibles los resultados del diseño procedimental).

2. ¿Hay errores mecanográficos?

3. ¿Se ha hecho un uso adecuado de las convenciones del lenguaje?

4. ¿Se han seguido los estándares de codificación para el estilo del lenguaje, los comentarios y los prólogos de los módulos?

5. ¿Hay comentarios incorrectos o ambiguos?

6. ¿Son apropiadas las declaraciones de tipos y de datos?

7. ¿Son correctas las constantes físicas?

8. ¿Se han vuelto a aplicar (si se requiere) todos los puntos de la lista de comprobaciones de la inspección del diseño?

Prueba del software. La prueba del software es una actividad de garantía de calidad por derecho propio. Por tanto, puede parecer redundante discutir las revisiones de la prueba. Sin embargo, se puede mejorar drásticamente la complejidad y la efectividad de la prueba validando críticamente cualquier plan o procedimiento de prueba que se haya establecido. Cabe aclarar que una de las mejores pruebas del software es permitir que el usuario final del software utilice el sistema y prueba cada una de las opciones del software.

Para el plan de prueba :

1. ¿Se han establecido correctamente los índices.?
2. ¿Se han identificado y secuenciado adecuadamente las principales fases de prueba?
3. ¿Se ha establecido un seguimiento de los criterios/requisitos de validación como parte del análisis de requisitos del software?
4. ¿Se han comprobado pronto las funciones importantes?
5. ¿Es consistente el plan de prueba con el plan global del proyecto?
6. ¿Se ha definido explícitamente un plan de tiempos para la prueba?
7. ¿Se han identificado y están disponibles los recursos y las herramientas para la prueba?
8. ¿Se ha establecido un mecanismo para registrar los resultados delas pruebas?

9. ¿Se han identificado los conductores y los resguardos y se ha planificado el trabajo para desarrollarlos?
10. ¿Se ha especificado la prueba de resistencia para el software?

Para el procedimiento de prueba :

1. ¿Se han especificado tanto pruebas de la caja negra como de la caja blanca?
2. ¿Se han probado todos los caminos lógicos independientes?
3. ¿Se han identificado y listado los casos de prueba junto con los resultados esperados?
4. ¿Se va a probar el manejo de errores?
5. ¿Se van a probar los valores límites?
6. ¿Se va a probar el rendimiento y las limitaciones temporales?
7. ¿Se ha especificado la variación aceptable respecto a los resultados esperados?

En función del cliente. La prueba del software es una actividad de garantía y es por ello que el cliente es un factor importante dentro de estas pruebas, por lo que es necesario considerar la respuesta a las siguientes preguntas :

1. ¿Se ha cumplido con lo que el cliente esperaba?

2. ¿Se han rebasado las expectativas del cliente sin aumento del costo?

El responder a estas preguntas no es fácil, considerando que la visión del cliente es distinta a la del ingeniero de software para la evaluación del sistema. Sólo el análisis respecto a las siguientes preguntas nos dará un mejor y mas objetivo panorama de lo que el cliente espera :

¿La respuesta del sistema es confiable?

¿El tiempo de respuesta es el correcto?

¿Es una herramienta útil en la toma de decisiones ?

¿Es una herramienta de fácil uso?

¿Los tiempos para el desarrollo de la aplicación fueron los estimados?

¿La aplicación incremento la productividad de la empresa? ¿Cómo?

¿Cómo fue el impacto en el personal de la empresa?

Además de las RTFs y de las listas de comprobaciones especificadas anteriormente, se pueden llevar a cabo revisiones (con sus correspondientes listas de comprobaciones) para asegurar la disposición de los mecanismos de servicio de campo para el producto de software, para evaluar la completitud y efectividad de la información, para asegurar la calidad de la documentación técnica y de usuario y para investigar la forma de aplicación y la disponibilidad de las herramientas de software.

Mantenimiento. Las listas de comprobaciones para la revisión del desarrollo del software son igualmente válidas para la fase de mantenimiento del software. Además de todas las preguntas propuestas en las listas de comprobaciones, se deben tener en cuenta las siguientes consideraciones especiales :

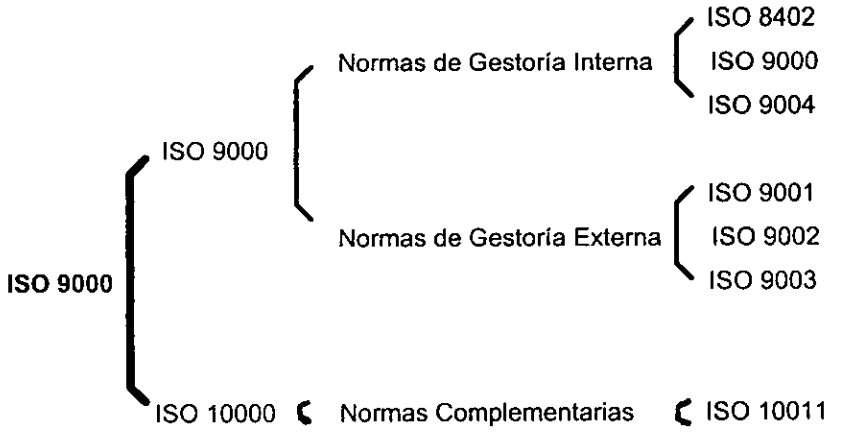
1. ¿Se han considerado los efectos laterales asociados con el cambio?
2. ¿Se ha documentado, evaluado y aprobado la petición de cambio?
3. ¿Se ha documentado el cambio, una vez hecho, e informado a las partes interesadas?
4. ¿Se han hecho RTFs adecuadas?

5. ¿Se ha hecho una revisión de aceptación final para garantizar que todo el software ha sido actualizado, probado y reemplazado adecuadamente?

- **ISO 9000 como Directriz**

La **ISO**.- Organización Internacional de Estándares (del inglés **International Standard Organization**), tiene su cede en Ginebra, Suiza y es una de las organizaciones que más se ha preocupado a nivel mundial por la estandarización. Existen otras organizaciones como la ANSI (American National Standars Institute) Instituto Nacional de Estándares y el IEEE (Institute of Electrical and Electronic Engineers) Instituto de Ingenieros Eléctricos y Electrónicos, que tienen una gran cantidad de estándares y de publicaciones sobre la estandarización de diversos aspectos relacionados con hardware y software.

La familia de normas ISO 9000 se divide como sigue :



ISO 8402:1994 Administración y aseguramiento de la calidad. Vocabulario
Es equivalente con la norma NMX-CC-001:1995

ISO 9000-1:1994 Administración y aseguramiento de la calidad. Parte 1. Directrices para la selección y uso.
Es equivalente con la norma NMX-002/1:1995

ISO 9001:1994 Sistemas de calidad-Modelo para asegurar la calidad en diseño, desarrollo, producción, instalación y servicio.
Es equivalente con la norma NMX-CC-003:1995

ISO 9002:1994 Sistemas de calidad-Modelo para asegurar la calidad en la producción, instalación y servicio.
Es equivalente con la norma NMX-CC-004:1995

ISO 9003:1994 Sistemas de calidad-Modelo para asegurar la calidad en la inspección y pruebas finales.
Es equivalente con la norma NMX-CC-005:1995

ISO 9004-1:1994 Administración de la calidad y elementos del sistema de calidad.- Directrices.
Es equivalente con la norma NMX-CC-006/1:1995

ISO 9004-2:1994 Administración de la calidad y elementos del sistema de calidad. Directrices para servicios.
Es equivalente con la norma NMX-CC-006/2:1995

ISO10011-1:1993 Directrices para auditar los sistemas de calidad.
Parte 1. Auditorias.
Es equivalente con la norma NMX-CC-007-1-1993

ISO10011-1:1993 Directrices para auditar los sistemas de calidad.
Parte 2. Admón. de los programas de auditoría.
Es equivalente con la norma NMX-CC-007-2-1993

ISO10011-2:1993 Criterios de calificación para los auditores de los sistemas de calidad.
Es equivalente con la norma NMX-CC-008 -1993

El estándar ISO 9000 ha sido adoptado por los sistemas estándares de la mayoría de países manufactureros alrededor del mundo. En la Comunidad Europea se conoce ahora como Euronorme EN29000-1987.

De manera básica, el ISO 9000 se refiere al aseguramiento de la calidad de las capacidades de la organización funcionales y consiste en documentar y estandarizar el proceso y representa en la actualidad la mejor práctica actual en administración y control cuando se aplica a la producción de software.

El sistema de calidad es necesario, toda vez que se debe de tener un plan básico para la calidad. Esto en realidad no sucede, solo es una forma de satisfacer de manera continua los requerimientos y expectativas de los clientes.

Un plan de calidad es un documento que establece los recursos de las prácticas especificas de calidad y actividades relevantes para un producto en particular, contratación de servicio o proyecto.

El sistema de calidad basado en ISO 9000 afecta a cualquiera asociado con la planeación, ventas, entrenamiento, suministro manufactura, inspección, pruebas, servicios al cliente e ingeniería; prácticamente a todos.

Los beneficios obtenidos con la implementación de un sistema de calidad basado en ISO 9000 pueden dividirse en beneficios para el cliente y para la compañía.

Beneficios para el cliente :

* Un nivel conocido de calidad que es definido y puede ser medido.

* Un nivel de servicio que ha sido y continua siendo auditado de manera independiente.

* Un desempeño que debe continuar mejorando.

* Un medio de seleccionar entre ofertas competitivas.

* Confidencialidad en los servicios proporcionados.

Beneficios para la compañía.

* Mejoras en la calidad a través de :

1) Incremento del conocimiento del personal.

2) Consistencia en productos y servicios.

* Los empleados entienden su rol y objetivos al tener un sistema de administración documentado.

* Una moral incrementada, al desarrollar un sentido de orgullo al alcanzar objetivos y proporcionar satisfacción al cliente.

* Productividad mejorada y ahorro en costos que hacen a la compañía más competitiva.

- Un sistema de administración eficiente.

ISO 9001

El documento ISO 9001 es un modelo genérico para un sistema de administración de calidad. El mismo documento puede usarse para construir un sistema de calidad en casi cualquier producto o servicio.

A continuación se comentan los 20 puntos que comprenden a ISO 9001:

1. Responsabilidad de la administración / política de calidad

Debe haber un programa definido para la calidad. La política debe ser comunicada y entendida a través de la compañía. La administración debe revisar de manera regular el sistema total de la calidad para asegurar su efectividad continua y conforme a ISO 9000. Debe haber administración de calidad efectiva con todas las responsabilidades definidas con claridad, por escrito, y suficientes recursos para hacer el trabajo.

Se debe tener administración con autoridad y responsabilidad definida(Organización).

2. Sistema de calidad

Todos los sistemas que directa o indirectamente afectan la calidad de nuestro producto y servicios deben ser documentados. Sin embargo, la documentación debe ser práctica, actual y controlada de manera efectiva.

También debe corresponder a lo que pasa en la realidad. Es esencial que todos sepan qué es lo que se supone deben hacer.

3. *Revisión del contrato*

Se debe asegurar que :

a) Se sabe cuales son los requerimientos del cliente
b) Que dichos requerimientos están documentados y revisados.
c) Todos los cambios a los requerimientos están resueltos.
d) Se tiene la capacidad de satisfacer los requerimientos.
e) Se está de acuerdo con el cliente.

- Los contratos y las órdenes deben ser revisados para garantizar que se continua satisfaciéndolos.
- Los clientes deben recibir los productos y servicios especificados

4. *Control del diseño*

Este requerimiento puede asegurar que el diseño terminado de los productos o de los servicios satisface aquellas necesidades especificadas por el cliente,
El diseño es el fundamento de la calidad. La secciones 3.2 y 3.3 tratan la necesidad de planear y revisar la estructura y la función del sistema en una etapa inicial.

5. *Control de documentos y datos*

La gente necesita saber qué se requiere que haga. Por lo tanto. las prácticas de trabajo deben estar documentadas, actualizadas, controladas y rápidamente disponibles. Los documentos obsoletos se deben retirar de uso con prontitud.
¿Estamos actualizados? ¿Todos utilizan la misma documentación?

6. *Adquisiciones*

Los materiales. partes y servicios que se compran a los proveedores deben

corresponder con el propósito, satisfacer los propios requerimientos de calidad y ser controlados. Es necesario que las especificaciones de compra estén escritas con claridad para que un proveedor sepa lo que se desea con exactitud,

El propósito es comprar sólo bienes y servicios que son de la calidad apropiada. Este requerimiento también incluye una estimación del vendedor y aseguramiento de subcontratos.

7. *Control de productos proporcionados por el cliente*

Los clientes pueden proveer materiales para utilizar en sus trabajos. Tales materiales deben ser almacenados de manera segura y es necesario usar algunos métodos para prevenir el deterioro o la pérdida.

8.- *Identificación y rastreabilidad del producto*

¿En cualquier momento es importante saber qué es cualquier ítem, a qué pertenece, en qué etapa está, qué etapas ha completado ya, qué fue usado para construirlo, de dónde vinieron las partes y quién hizo qué?

Una vez que un ítem ha sido construido, es necesario que sepamos hacia dónde se dirige. Cuando sea apropiado, debe haber un seguimiento tanto hacia atrás como hacia delante.

Si un elemento está equivocado en cualquier momento, su historia nos dice qué hacer para prevenir fallas futuras. Los sistemas grandes de software ,Tienen muchos componentes, de ahí que sea esencial ser capaz de agrupar los correctos.

9.- *Control del proceso*

La producción e instalación debe ocurrir bajo condiciones controladas. Los empleados deben saber qué hacer y qué estándares se requieren.

Es necesario que existan controles asegurar que sólo se proporcionan productos

de buena calidad. La operación puede ser compleja, de ahí que puede requerir verificaciones especiales, instrucciones adicionales, entrenamiento y calificaciones específicas. Las instrucciones adecuadas de trabajo así como los estándares a satisfacer se deberán escribir para asegurar que el trabajo se hace en forma correcta con las herramientas, partes, métodos y especificaciones apropiadas.

La calidad consistente requiere de procesos controlados. Siempre es necesaria una instrucción de trabajo: la ausencia de una afecta la calidad.
Las instrucciones de trabajo deben ser fáciles de entender, prácticas. cortas y probadas

10. Inspección y prueba

Los productos deben ser inspeccionados o verificados, y pasar el criterio de aceptación al momento de ser recibidos, trabajados y previamente a la distribución.
Todos los productos que no correspondan a esos estándares deberán segregarse para que una acción correctiva pueda llevarse a cabo. Es indispensable guardar los registros a fin de inspeccionar el trabajo efectuado en cada etapa.
Asegurar que el producto cumple todas las especificaciones.

11.- Control de equipo de inspección, medición y prueba.

El equipo utilizado para la verificación o prueba de un producto tiene que ser capaz de desempeñar en forma correcta estas funciones, también se requiere que:

- Todo el equipo de prueba y medición esté identificado.
- Los registros muestren la frecuencia de calibración.
- Existan procedimientos de calibración de acuerdo a estándares.

12. Estado de inspección y prueba.

Distinguir entre inspeccionado y no inspeccionado (bueno y malo).

13. Control de productos no satisfactorios.

La reducción del desecho y la necesidad de volver a trabajar incrementa la eficiencia.

¿Tenemos software desechado?

14. Acción correctivo y preventiva.

Todos los procedimientos, procesos y productos necesitan ser monitoreados y revisados para asegurar la entrega constante de calidad. Cualquier estándar no satisfecho debe ser documentado, analizado, identificar la causa y tomar acciones correctivas y preventivas.

15. Manejo, almacenamiento, empaque, conservación y entrega.

Los productos deben ser manejados con mucho cuidado a fin de prevenir daños. Deben ser almacenados en áreas seguras y controladas. Es necesario que todo el paquetaje esté en un estándar adecuado y, por lo tanto, prevenir el daño o deterioro de los productos mientras se encuentren en almacén o se entregan.

16.- Control de registros de calidad.

Los registros de calidad deben mantenerse para que puedan demostrar nuestro nivel de alcance contra los estándares requeridos y, de hecho probar la efectividad del sistema de calidad.

17.- Auditorías de calidad internas.

Los auditores del aseguramiento de la calidad interna llevan a cabo garantías y pruebas imparciales independientes para verificar si tanto los procesos como los procedimientos siguen siendo pertinentes, efectivos y satisfacen los requerimientos.

18. Entrenamiento.

Para que los empleados desempeñen las tareas de manera satisfactoria y cumplan con los estándares de los requerimientos, se deben de capacitar en los métodos y técnicas correctos a utilizar.

Se debe documentar el entrenamiento dentro y fuera del trabajo así como la experiencia derivada de ello. Debe haber un plan de cualquier entrenamiento futuro requerido.

19. Servicio

Donde el servicio de equipo es un requerimiento, debe estar bien especificado, llevado a cabo por una capacitación adecuada al personal y dentro de las escalas de tiempo acordadas.

Un servicio bien hecho, a tiempo, por un equipo de trabajo capacitado de manera adecuada provee clientes satisfechos y felices.

20. Técnicas estadísticas

Se puede revisar o verificar la aceptación de los productos y procesos en una base ejemplo, pero el ejemplo debe ser representativo del universo. Los registros deben guardarse para identificar las tendencias y prevenir la asistencia en caso necesario.

Aún no se tiene ninguna métrica aceptada a nivel universal que pueda ser utilizada para predecir o cuantificar el softwarei sin embargo podemos usar los métodos estadísticos básicos e intermedios en el análisis de datos obtenidos de las características propias de un software, para utilizarlos en el futuro como punto de comparación.

Estos 20 puntos explican lo que tiene que controlarse para instalar un sistema de calidad basado en el estándar ISO 9000.

♦ METRICAS DE CALIDAD DEL SOFTWARE

Anteriormente en este capítulo hemos visto los factores cualitativos que permiten "medir" la calidad del software. A veces, cuando intentamos obtener medidas precisas de la calidad del software acabamos frustrados por la naturaleza subjetiva de esa actividad. Cavano y McCall [CAV78] tratan esta situación :

La determinación de la calidad es un factor clave en cualquier suceso cotidiano - concursos de caja de vino, sucesos deportivos [p. ej.: gimnasia], concursos de talento, etc. En estas situaciones, la calidad se juzga de la manera más básica y directa; por comparación entre los objetos, bajo idénticas condiciones y con conceptos previamente determinados.

El vino puede ser juzgado de acuerdo con su gama, color, bouquet, sabor, etc. Sin embargo, este tipo de juicio es muy subjetivo, para que tenga algún valor debe ser establecido por un experto.

La subjetividad y la especialización también se pueden aplicar a la determinación de la calidad del software. Para ayudar a resolver este problema, se necesita una definición más precisa de la calidad del software, así como una forma de obtener medidas cuantitativas de la calidad del software, con el fin de poder llevar a cabo un análisis objetivo. Como no existe tal cosa como conocimiento absoluto, no se puede esperar medir la calidad del software de forma exacta, ya que cada medida es parcialmente imperfecta. Jacob Bronowski describe esta paradoja del conocimiento de la siguiente forma :

"Año tras año conseguimos instrumentos más precisos con los que observar con más detalle la naturaleza y cuando miramos las observaciones no estamos dispuestos a admitir que sean lo suficientemente precisas y sentimos que son tan inciertas como siempre".

En esta sección veremos un conjunto de métricas del software que se pueden aplicar para garantizar cuantitativamente la calidad del software. En todos los casos, las métricas representan medidas indirectas, es decir, nunca medimos realmente la calidad, sino algunas de sus manifestaciones. El factor que lo complica es la relación precisa en la variable que es medida y la calidad del software.

Indices de calidad del software

El US Air Force Systems command ha desarrollado una serie de indicadores de calidad del software basados en las características de diseño medibles para un programa de computadora. Con conceptos similares a los propuestos por el estándar del IEEE 982.1-1988, en la Air Force utilizan información obtenida a partir del diseño arquitectónica y de datos, para obtener un índice de calidad de la estructura del diseño (ICED), que va de 0 a 1. Para calcular el ICED se tienen que averiguar los siguientes valores:

S_1 = número total de módulos definidos en la arquitectura del programa

S_2 = número de módulos cuya correcta función depende de la fuente de los datos de entrada o que produce datos que se usan en cualquier parte (en general, los módulos de control, entre otros, no se van a considerar como parte de S_2).

S_3 = número de módulos cuya correcta función depende del procesamiento previo.

S_4 = número de elementos de una base de datos (incluye los objetos de datos y todos los atributos que definen objetos).

S_5 = número total de elementos de base de datos únicos.

S_6 = número de segmentos de base de datos (registros diferentes u objetos individuales)

S_7 = número de módulos con una sola entrada y una sola salida (el procesamiento de excepciones no se considera como una salida múltiple).

Una vez determinados los valores S_1 a S_7 para un programa de computadora, se pueden calcular los siguientes valores intermedios :

Estructura del programa: D_1 que se define de la siguiente forma : Si el diseño arquitectónico se desarrolló usando un método característico (p. ej.: el diseño orientado al flujo de datos o diseño orientado a los objetos), entonces $D_1 = 1$; en caso contrario $D_1 = 0$.

independencia de módulos : $D_2 = 1 - (S_2/ S_1)$

Módulos no dependientes del procesamiento previo : $D_3 = 1 - (S_3/ S_1)$

Tamaño de la base de datos : $D_4 = 1 - (S_5/S_4)$

Compartimentalización de la base de datos : $D_5 = 1 - (S_6/ S_4)$

Característica de entrada/salida del módulo : $D_6 = 1 - (S_7/ S_1)$

Habiendo determinado esos valores intermedios, el ICED se calcula de la siguiente manera:

$$\mathbf{ICED = \sum P_i D_i}$$

donde i varía de 1 a 6, P_i es el peso relativo de la importancia de cada uno de los valores intermedios y $\sum P_i = 1$ (si todos los D_i tienen el mismo peso, entonces $P_i = 0.167$).

Se puede determinar el valor del ICED para anteriores diseños y compararlo con un diseño que actualmente esté en desarrollo. Si el valor del ICED es significativamente menor que la media, significará que se vá a requerir un posterior trabajo de diseño y de revisión. Igualmente, si hay que hacer cambios importantes en un diseño existente, se puede calcular el efecto de esos cambios en el ICED.

El estándar del IEEE 982.1-1988 sugiere un *índice de madurez del software* (IMS), que proporciona una indicación de la estabilidad de un producto de software (basada en los cambios que se producen en cada versión de producto). Se determina la siguiente información :

M_T = número de módulos en la versión actual.

F_m = número de módulos en la versión actual que han sido modificados.

F_a = número de módulos en la versión actual que han sido añadidos

F_e = número de módulos de la versión anterior que se han eliminado en la versión actual.

El índice de madurez del software se calcula de la siguiente forma:

$$\mathbf{IMS = [\ MT\ (Fa + Fm + Fe)\ /\ MT\ ]}$$

A medida que el IMS se aproxima a 1, el producto comienza a estabilizarse. El IMS también se puede utilizar como métrica para la planificación de actividades del mantenimiento del software. El tiempo medio para producir una versión de un producto de software puede tener correlación con el IMS y se pueden desarrollar modelos empíricos para el esfuerzo de mantenimiento.

Proceso limpio

La verificación formal de programas (pruebas de corrección) y la SQA estadística se han combinado en una técnica que mejora la calidad del producto software. Denominado proceso limpio o ingeniería del software limpia, es descrito por MillS[2] de la siguiente forma :

Con el proceso limpio, se puede hacer ingeniería de software bajo un control de calidad estadístico. Igual que con el desarrollo de hardware limpio, la mayor prioridad del proceso es la prevención de los defectos, más que la eliminación de defectos (por supuesto, se debe eliminar cualquier defecto no previsto).
Esta primera prioridad se consigue mediante la verificación matemática [pruebas de corrección] en lugar de la depuración de programas que prepara el software para la prueba del sistema.

La siguiente prioridad es proporcionar una certificación estadística válida de la calidad del software
La medida de la calidad viene dada por el tiempo medio hasta que se produce el fallo...

[2] Mills, H.D., M. Dyer, and R. C. Linger "Clearroom Software Enginnering" September 1987

- ***FIABILIDAD DEL SOFTWARE***

No hay duda de que la fiabilidad de un programa de computadora es un elemento importante de su calidad general. Si un programa falla frecuentemente en su funcionamiento, no importa si el resto de los factores de calidad son aceptables

La fiabilidad del software, a diferencia de otros factores de calidad, puede ser medida o estimada mediante datos históricos o de desarrollo. La fiabilidad del software se define en términos estadísticos como "la probabilidad de operación libre de fallos de un programa de computadora en un entorno determinado y durante un tiempo especifico" [MUS87]. Por ejemplo, un programa **x** puede tener una fiabilidad estimada de 0.96 durante un intervalo de procesamiento de ocho horas. En otras palabras, si se fuera a ejecutar el programa **x** 100 veces necesitando ocho horas de tiempo de procesamiento (tiempo de ejecución), lo probable es que funcione correctamente (sin fallos) 96 de cada 100 veces.

Siempre que se habla de fiabilidad, surge una pregunta fundamental : ¿qué se entiende por el término fallo? En el contexto de cualquier disquisición sobre calidad y fiabilidad del software, el fallo es cualquier falta de concordancia con los requisitos del software. Incluso en esta definición existen grados. Los fallos pueden ser simplemente desconcertantes o ser catastróficos. Puede que un fallo Corregido en segundos mientras que otro lleve semanas o incluso meses. Para complicar más las cosas, la corrección de un fallo puede llevar a la introducción de otros errores que, finalmente, lleven a mas fallos.

Medidas de fiabilidad y de disponibilidad

Los primeros trabajos sobre fiabilidad intentaron extrapolar las matemáticas de la teoría de fiabilidad del hardware a la predicción de la fiabilidad del software. La mayoría de los módulos de fiabilidad relativos al hardware van más orientados a

los fallos debido al desajuste que a los fallos debidos a defectos de diseño. En el hardware, son más probables los fallos debidos al desgaste físico (p. ej.: el defecto de la temperatura, de la corrosión, de los golpes) que los fallos relativos al diseño. Desgraciadamente, para el software lo que ocurre es lo contrario. De hecho, todos los fallos del software, se producen por problemas de diseño o de implementación; el desajuste no entra en este panorama.

Todavía se debate sobre la relación entre los conceptos clave de la fiabilidad del hardware y su aplicabilidad al software. Aunque todavía falta por establecer un nexo irrefutable, merece la pena considerar unos cuantos conceptos que conciernen a ambos elementos de los sistemas.

Considerando un sistema basado en computadora, una sencilla medida de la fiabilidad es el *tiempo medio entre fallos* (TMEF), donde :

TMEF = TMDF + TMDR

Donde:
TMDF = Tiempo medio de fallo
FMDR = Tiempo medio de reparación,

Muchos investigadores argumentan que e TMEF es, con mucho, una medida más útil que los defectos/KLDC. Sencillamente, el usuario final se enfrenta a los fallos, no al número total de errores. Como cada error de un programa no tiene La misma tasa de fallo, la cuenta total de errores no es una buena indicación de la fiabilidad de un sistema. Por ejemplo, consideremos un programa que ha estado funcionando durante 14 meses. Muchos de los errores del programa pueden pasar desapercibidos durante décadas antes de que se detecten. El TMEF de esos errores puede ser 50 e incluso de 100 años. Otros errores, aunque no se hayan descubierto aún pueden tener una tasa de fallo de 18 o 24 meses.

Incluso aunque se eliminen todos los errores de la primera categoría (los que tienen un gran TMEF), el impacto sobre la fiabilidad será muy escaso.

Además de una medida de la fiabilidad debemos obtener una medida de la *disponibilidad*. La disponibilidad del software es la probabilidad de que un programa funcione de acuerdo con los requisitos en un momento dado, se define como :

Disponibilidad = [TMDF / (TMDF + TMDR)] x 100

La medida de fiabilidad TMEF es igualmente sensible al TMDF que al TMDR La medida de disponibilidad es algo más sensible al TMDR, una medida indirecta de la facilidad de mantenimiento del software.

Modelos de fiabilidad del software

En un extenso tratado sobre fiabilidad del software, Musa y sus colegas[MUS87] describen los modelos de fiabilidad del software de la siguiente forma:

Para Modelizar la fiabilidad del software se deben considerar primero los principales factores que afectan : introducción de fallos, eliminación de fallos y entorno. La introducción de fallos depende principalmente de las caracteristicas del código desarrollado (código creado o modificado para la aplicación) y de las características del proceso de desarrollo. La característica del código más significativa es el tamaño. Entre las características del proceso de desarrollo se encuentran las tecnologías y las herramientas de ingeniería del software usadas, y el nivel de experiencia del personal. Se puede desarrollar código para añadir posibilidades o para eliminar fallos. La eliminación de fallos depende del tiempo, del perfil operativo y de la calidad de la actividad de reparación. El entorno depende del perfil operativo. Como algunos de los anteriores factores son de

naturaleza probabilística y se dan en el tiempo, los módulos de fiabilidad del software generalmente se formulan en términos de procesos aleatorios.

Los modelos de fiabilidad del software entran en dos grandes categorías : modelos que predicen la fiabilidad como una función cronológica del tiempo (calendario) y (2) modelos que predicen la fiabilidad como una función del tiempo de procesamiento transcurrido (tiempo de ejecución de CPU). Musa y cois sugieren que los modelos de fiabilidad del software basados en el tiempo de procesamiento transcurrido (tiempo de ejecución) dan los mejores resultados globales.

Los modelos que se han obtenido de los trabajos sobre fiabilidad del hardware hacen las siguientes suposiciones : (1) el tiempo de depuración entre ocurrencias de error tiene una distribución exponencial para una tasa de ocurrencia de error proporcional al número de errores restantes ; (2) cada error es inmediatamente corregido, decrementando el número total de errores en uno ; (3) la tasa de fallo entre errores es constante. La validez de cada una de estas suposiciones es cuestionable. Por ejemplo, las correcciones de un error puede inadvertidamente introducir otros errores en el software, invalidando la segunda suposición.

Otra clase de modelos de fiabilidad se basa en las características internas de un programa y calcula el número predecible de errores que existen en el software. Estos modelos, basados en las relaciones cuantitativas obtenidas como una función de las medidas de complejidad del software, relacionan las atributos específicos del diseño u orientados al código de un programa (p. ej.: número de operandos y de operadores o la complejidad ciclomática) como una estimación del número inicial de errores esperados en un programa dado.

Los modelos de diseminación se usan para obtener una indicación de la fiabilidad del software o, de forma más práctica, para medir el "poder de detección de errores" de un conjunto de casos de prueba. Se disemina un programa de forma

aleatoria con cierto número de errores de "calibración" conocidos. Se prueba el programa (mediante casos de prueba). Se puede relacionar la probabilidad de encontrar J errores reales de una población total de j errores (desconocidos) con la probabilidad de encontrar k errores diseminados de entre todos los k errores introducidos en el código.

Se han propuesto modelos escolásticos mucho más sofisticados para la fiabilidad del software. Para aquellos lectores que quieran estudiar tales modelos en profundidad, se sugiere un conjunto de criterios de comparación y de evaluación :

Validez predictiva : la posibilidad de que el modelo prediga el comportamiento de fallo futuro basándose en los datos obtenidos de las fases de prueba y de operación.

Capacidad : la posibilidad de que el modelo genere datos que puedan ser fácilmente aplicados a los esfuerzos de desarrollo de software industriales.

Calidad de las suposiciones : la plausibilidad de las suposiciones en las que se basan los fundamentos matemáticos del modelo y el grado de degradación del modelo cuando se llega a los límites de esas suposiciones.

Aplicabilidad : el grado en que se puede aplicar un modelo de fiabilidad en diferentes terrenos y tipos de aplicación del software.

Simplicidad : el grado en que el conjunto de datos que soportan el modelo es directo; el grado en que el enfoque y las matemáticas son intuitivos; el grado en que se puede automatizar el enfoque general.

Una discusión de los modelos a los que se pueden aplicar estos criterios requiere experiencia en estadística y probabilidad y se deja para los libros dedicados a la fiabilidad del software.

Seguridad del software

Cada vez se hace más presente e indispensable el uso de las computadoras en procesos críticos de seguridad, tales como uso en la supervisión de reactores nucleares, control de vuelos, sistemas de defensa y procesos industriales a gran escala. Aunque la probabilidad de fallo de un sistema bien construido es pequeña, un fallo no detectado en un sistema de control o de supervisión basado en computadora podría originar un enorme daño económico o, lo que es peor daños humanos importantes o pérdida de vidas. Pero los beneficios de costo, funcionalidad y supervisión basado en computadora, normalmente, pesan mas que el riesgo. Actualmente el hardware y el software se utilizan para regular y controlar sistemas críticos de seguridad. Leveson [LEV86] discute el impacto del software en sistemas críticos de seguridad, diciendo :

"Antes de que se usara software en sistemas críticos de seguridad estos se controlaban mediante dispositivos electrónicos y mecánicos (no programables). Las técnicas de seguridad de sistemas se diseñaban frente a fallos aleatorios en esos dispositivos (no programables). Los errores humanos de diseño no se consideraban porque se suponía que todos los defectos por los errores humanos se podían evitar o eliminar completamente antes de su distribución y funcionamiento.

Cuando se utiliza el software como parte del sistema de control, la complejidad puede aumentar en un orden de magnitud o más. Los defectos sutiles de diseño producidos por un error humano - algo que se puede descubrir y eliminar en el control convencional basado en el hardware - llegan a ser mucho más difíciles de descubrir cuando se utiliza el software.

La seguridad del software es una actividad de garantía de calidad del software que se centra en la identificación y evaluación de los riesgos potenciales que pueden producir un impacto negativo en el software y hacer que falle el sistema completo.

Si se pueden identificar pronto los riesgos en el proceso de ingeniería del software se pueden especificar pronto las características de diseño de software que podrán eliminar o controlar los riesgos potenciales.

Como parte de la seguridad del software, se puede dirigir un proceso de análisis y modelización. Inicialmente, se identifican los riesgos y se clasifican por su importancia y su grado de riesgo. Por ejemplo, algunos de los riesgos asociados con el control basado en computadora del crucero de un automóvil podrían ser :

- Produce una aceleración incontrolado que no se puede detener
- No responde al abatimiento del pedal de freno (al soltar)
- No responde cuando se activa el contacto
- Pierde o gana velocidad lentamente

Cuando se han identificado estos riesgos del sistema, se utilizan técnicas de análisis para asignar su gravedad y su probabilidad de ocurrencia. Para que sea efectivo, se tiene que analizar el software en el contexto del sistema completo. Por ejemplo, puede que un sutil error en la entrada del usuario (las personas son componentes del sistema) se magnifique por un fallo del software que produce los datos de control que actúan de forma inadecuada sobre un dispositivo mecánico. Si se da una serie de condiciones externas del entorno (y sólo si se da), la situación inadecuada del dispositivo mecánico producirá un fallo desastroso. Se pueden usar técnicas de análisis, como el análisis del árbol de fallos, la lógica de tiempo real o los modelos de redes de petri para predecir la cadena de sucesos que pueden producir los riesgos y la probabilidad de que se dé cada uno de los sucesos que componen la cadena.

El análisis del árbol de fallos construye un modelo gráfico de las combinaciones secuenciales y concurrentes de los sucesos que pueden conducir a un suceso o estado del sistema peligroso. Mediante un árbol de fallos bien desarrollado, es posible observar las consecuencias de una secuencia de fallos interrelacionados que ocurren en diferentes componentes del sistema. La lógica de tiempo real (LTR) construye un modelo del sistema mediante la especificación de los sucesos y las acciones correspondientes. El modelo suceso-acción se puede analizar mediante operaciones lógicas para probar las valoraciones de seguridad de los componentes del sistema y su temporización. Se pueden usar los modelos de redes de Petri para determinar los riesgos más peligrosos.

Cuando se han identificado y analizado los riesgos, se pueden especificar requisitos del software relacionados con la seguridad. La especificación puede contener una lista de sucesos no deseables y las respuestas del sistema deseadas dichos sucesos. Así, se indica el papel del software en la gestión de los sucesos no deseables.

En una sección anterior se trató el papel del análisis de la fiabilidad del software Aunque la fiabilidad y la seguridad del software están bastante relacionadas, es importante entender la sutil diferencia que existe entre ellas. La fiabilidad del software utiliza el análisis estadístico para determinar la probabilidad de que pueda ocurrir un fallo del software. Sin embargo, la ocurrencia de un fallo no lleva necesariamente a un riesgo o a un accidente. La seguridad del software examina los modos según los cuales los fallos producen condiciones que pueden llevar a accidentes. Es decir, los fallos no se consideran en vacío, sino que se evalúan en el contexto de un completo sistema basado en computadora.

♦ *UN ENFOQUE PARA LA GARANTIA DE CALIDAD DEL SOFTWARE*

Aunque pocos profesionales pondrían en duda la necesidad de calidad en el software, muchos no están interesados en establecer funciones de SQA formales. Las razones de esta aparente contradicción son muchas : (1) los responsables del desarrollo se resisten a hacer frente a los costos extras inmediatos ; (2) los profesionales creen que ya están haciendo todo lo que hay que hacer ; (3) nadie sabe donde situar esa función dentro de la organización ; (4) todos quieren evitar el papeleo que la SQA tiende a introducir en el proceso de ingeniería del software.

En esta sección presentamos un breve tratamiento de los aspectos más importantes que conciernen a la institucionalización de las actividades garantía de calidad del software. Para una presentación más detallada.

Necesidad de SQA

Todas las organizaciones de desarrollo de software tienen algún mecanismo de garantía de calidad. En el nivel inferior de la escala, la calidad es responsabilidad únicamente del individuo que deba crear, revisar y probar el software a cualquier nivel de conformismo. En el nivel superior de la escala, existe un grupo de SQA que carga con la responsabilidad de establecer estándares y procedimientos para conseguir la calidad del software y asegurar que se sigue cada uno de ellos. La verdadera cuestión que surge para cualquier organización de desarrollo de software es : ¿dónde nos situamos en la escala?

Antes de institucionalizar procedimientos formales de garantía de calidad, una organización de desarrollo de software debe adoptar procedimientos, métodos y herramientas de ingeniería del software. Esta metodología, combinada con un

paradigma efectivo para el desarrollo de software, puede hacer mucho por mejorar la calidad de todo el software desarrollado por la organización.

El primer paso a dar como parte de un decidido esfuerzo por institucionalizar los procedimientos de garantía de calidad del software es una auditoría SQA/GCS. El "estado" actual de la garantía de calidad del software y de la gestión de configuraciones del software se evalúa examinado los siguientes puntos :

Principios. ¿Qué principios, estándares y procedimientos existen actualmente para todas las fases de desarrollo del software? ¿Son forzosos? ¿Existe algún principio específico (de gestión) para la SQA? ¿Se aplican los principios tanto a las actividades de desarrollo como a las de mantenimiento?

Organización. ¿Dónde reside actualmente la ingeniería del sottware en el esquema organizativo? ¿Dónde reside la garantía de calidad?

Interfaces funcionales. ¿Cuál es la relación actual entre las funciones de garantía de calidad y las otras asociadas? ¿Cómo interactúa la SQA en la gente que realiza las revisiones técnicas formales, con la gestión de configuraciones y con las actividades de prueba?

Una vez que se ha respondido a estas preguntas, se identifican las debilidades y las posibilidades. Si resulta evidente la necesidad de SQA, se lleva adelante una cuidadosa evaluación de los pros y los contras.

En el lado positivo, la SQA ofrece los siguientes beneficios :

1.El software tendrá menos defectos latentes, resultando un menor esfuerzo y un menor tiempo durante la prueba y el mantenimiento.

2.Se dará una mayo fiabilidad y por tanto, una mayor satisfacción del cliente.

3.Se podrán reducir los costos de mantenimiento (un porcentaje sustancial de los costos totales del software) y...

4.El costo del ciclo de vida total del software disminuirá. Como dice Crosby la calidad es gratis

En el lado negativo, la SQA puede ser problemática por las siguientes razones :

Es difícil de institucionalizar en organizaciones pequeñas, en las que no están disponibles los recursos necesarios para llevar a cabo esas actividades.

Representa un cambio cultural, y el cambio nunca es fácil.

Requiere un gasto que, de otro modo, nunca se hubiera destinado explícitamente a la ingeniería del software o a la garantía de calidad.

En un nivel fundamental, la SQA es efectiva en costo si

$$C3 > C1 + C2$$

donde C3 es el costo de los errores que aparecen sin un programa de SQA, C1 es el costo del propio programa de SQA y C2 es el costo de los errores que no se encuentran con las actividades de SQA. Sin embargo es importante resaltar que en un análisis más minucioso habría que considerar también las reducciones en los costos de la prueba y de la integración, el menor número de cambios en las primeras versiones, los menores costos de mantenimiento y la mejora en satisfacción del cliente. En general, la SQA evoluciona como parte de un esfuerzo general de gestión dirigido a mejorar la calidad.

CAPITULO III : INGENIERIA DE SOFTWARE ASISTIDA POR COMPUTADORA (CASE)

La tecnología para el desarrollo de sistemas de información asistido por computadoras (CASE por las siglas de su nombre en inglés **Computer Aided Software Engineering**), representa una filosofía completa para modelar a los negocios, sus actividades y el desarrollo de sistemas de información. Involucra el uso d computadoras como herramientas de desarrollo para construir modelos que describen al negocio, el ambiente del negocio y la planeación corporativa, así como para documentar el desarrollo de sistema computarizados desde la planeación hasta la implementación.

La completa imagen para la filosofía CASE prescribe que la especificación de los planes corporativos, el diseño y desarrollo de los sistemas se vuelvan totalmente integrados. Esto ocurre al compartir especificaciones para las tres funciones de planeación corporativa, análisis y diseño de sistemas, y desarrollo de sistemas mediante los componentes CASE.
La tecnología CASE esta sustituyendo al papel y lápiz con la computadora para transformar el desarrollo del software en un proceso automatizado. Definido simplemente, CASE es la automatización del software. La idea básica subyacente a la tecnología CASE es la de proveer de un conjunto de herramientas bien integradas, que ahorran trabajo y que automatizan el desarrollo de software.

CASE es una combinación de herramientas de software y de metodologías estructuradas de desarrollo de software. Las herramientas automatizan el proceso de software y las metodologías definen el proceso a ser automatizado. La tecnología CASE se enfoca en la productividad de los desarrolladores de sistemas de software de negocios, de tiempo real y científicos. Con el uso de las computadoras personales, estaciones de trabajo, redes y herramientas CASE, los desarrolladores de software pueden trabajar desde un ambiente dedicado

altamente responsivo para desarrollar y dar mantenimiento a sistemas de software.

Las herramientas CASE son una nueva clase de herramientas de software orientadas a gráficos, basados en computadoras. Conforme la noción de CASE ha evolucionado en los últimos años, la definición de una herramienta CASE se ha ampliado de ser simplemente herramientas de análisis y documentación de sistemas a incluir una función completa proveyendo soporte automatizado al ciclo de vida de sistemas completos.

La definición más amplia de una herramienta CASE es: "toda herramienta de software que provee asistencia automatizada para el desarrollo, mantenimiento y administración de proyectos de software"

Pero para calificar como una herramienta CASE bien nacida, hoy en día una herramienta de software tiene que exhibir características adicionales. Las herramientas CASE usan gráficos poderosos para
describir y documentar a los sistemas de software, mejorando de manera inherente la interface con el usuario. También se encuentran integradas, haciendo fácil el pasar datos de herramienta a herramienta, capturando la información de los sistemas de software en un repositorio computarizado, en donde puede ser compartida por los desarrolladores de software, usada como base :ara la producción automatizada del software y reutilizada en sistemas futuros.

Con tantas herramientas CASE en el mercado, la correcta elección requiere de alguna consideración. Ninguna herramienta puede proporcionar un soporte completamente automatizado para el desarrollo y mantenimiento de todo tipo de sistema de software. Diferentes herramientas CASE corren en diferentes tipos de equipos, se especializan en el desarrollo y mantenimiento de diferentes sistemas de software y automatizan diferentes tareas de software.

La consideración fundamental de selección es lo que hace la herramienta. Categorías de Herramientas CASE explican y diferencian las funciones realizadas por los diferentes tipos de herramientas CASE.

♦ PRINCIPIOS DE LA TECNOLOGÍA CASE

Podemos mencionar a continuación las ideas básicas por las que la tecnología CASE logra ser un medio alterno para lograr la maximización de los recursos de software.

* LAS TECNICAS DE DISEÑO NECESITAN DE AUTOMATIZACION.

El cerebro humano cuenta con una capacidad limitada para manejar al detalle la complejidad, el rigor matemático y la verificación sin cometer errores. Al utilizar adecuadamente la computadora, los diseñadores pueden rebasar los confines del cerebro humano en un menor tiempo y con un grado de error prácticamente nulo.

* EL DISEÑO ASISTIDO POR COMPUTADORA REQUIERE DE GRAFICAS. Todo análisis y diseño estructurado requieren gráficas, en este sentido los diagramas se tornan cada vez más complejos y el cambiarlos o modificados consume tiempo. Una herramienta computarizada debe facilitar la construcción, edición, refinamiento, expansión y extensión de subconjuntos de los diagramas del diseño, de manera que la complejidad se haga más fácil de manejar.

* LA HERRAMIENTA COMPUTARIZADA DEBE GUIAR AL DISEÑADOR. Las herramientas CASE deben incorporar metodologías y guiar al diseñador a través de bases que reflejen un buen diseño. La herramienta deberá solicitar información al diseñador de manera que tenga un conjunto completo de información a partir del cual generar el código.

* SE DEBERA EVITAR LA PROGRAMACION LO MAS POSIBLE. Los diagramas de diseño deben ser divisibles los cuales a su vez sucesivamente proyectaran más detalle hasta que se pueda generar código de ellos automáticamente. Los reportes, las gráficas, las bases de datos, los diagramas, la actualización de los datos, etc. deben ser especificables para la generación automática de código.

* SE DEBEN EMPLEAR CONSTRUCCIONES QUE CONDUZCAN A UN DISEÑO AXIOMATICAMENTE DEMOSTRABLE. Se debe hacer el máximo uso de construcciones que conduzcan a una automatización matemáticamente rigurosa del diseño.

* LOS LENGUAJES DEBEN SER CONFECCIONADOS CONFORME A LAS TECNICAS DE DISEÑO. Los nuevos lenguajes de cuarta generación están evolucionando rápidamente, estos deben emplear construcciones apropiadas de tal manera que se reflejen las mejores técnicas para el diseño automatizado, en el lenguaje de computadora.

Beneficios de la tecnología CASE

Entre las principales ventajas de las que podamos gozar al hacer un uso adecuado de esta tecnología tenemos las siguientes:

- Hace prácticas a las técnicas estructuradas.
- Refuerza a la ingeniería de software / información.
- Mejora la calidad del software mediante el chequeo automático.
- Hace práctico el desarrollo de prototipos.

- Simplifica el mantenimiento de programas.

- Acelera el proceso de desarrollo.

- Libera a los desarrolladores para que se enfoquen en la parte creativa del desarrollo de software.

- Alienta el desarrollo evolucional e incremental.

- Permite la reutilización de componentes de software.

- No requiere de un programador experto para su uso (son herramientas intuitivas).

♦ BLOQUES QUE COMPONEN EL CASE

En relación con la ingeniería de software asistida por computadora podemos considerar que puede ser tan simple como una única herramienta que permita desarrollar una actividad específica, o tan compleja como un "entorno" que integre distintas herramientas, una base de datos, gente, hardware, una red, sistemas operativos, estándares y muchos otros componentes.
Cada bloque constituye la base siguiente con las herramientas situadas en la cima de la pila. Es interesante ver que el fundamento para un CASE efectivo tiene poco que ver con las herramientas de ingeniería de software en sí mismos. Los buenos entornos de ingeniería de software se construyen sobre una arquitectura de entorno que engloba los correspondientes sistemas de software y de hardwarei además, la arquitectura de entorno debe considerar los patrones de trabajo humanos que se aplican durante el proceso de ingeniería de software.

En la siguiente Figura, se representan los bloques que componen al CASE.

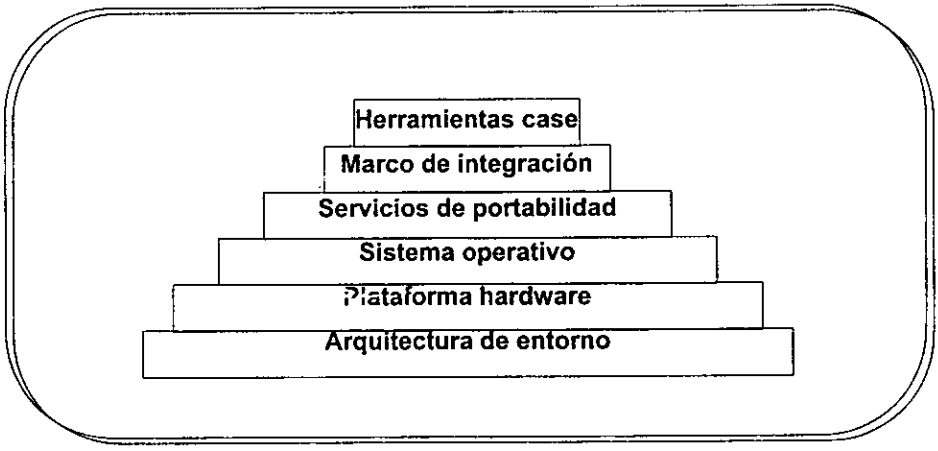


Figura 3.1 Bloques constitutivos del CASE

Hoy en día la tendencia en el desarrollo de software está lejos de las grandes computadoras y enfocada hacia las estaciones de trabajo como plataformas de ingeniería de software. Las estaciones de trabajo individuales se interconectan mediante redes para que los ingenieros de software puedan comunicarse de forma efectiva. La base de datos de proyectos deberá estar disponible a través de un servidor de archivos en red que es accesible desde todas las estaciones de trabajo. Un sistema operativo que gestiona el hardware de la red y las herramientas mantiene todo el entorno unido.

La arquitectura de entorno, compuesta por la plataforma de hardware y el soporte del sistema operativo (incluida la red y la gestión de la base de datos) constituyen la base del CASE.

Pero el entorno CASE, en sí mismo, necesita otros componentes. Un conjunto de servicios de portabilidad constituyen un puente entre las herramientas CASE y su marco de integración y la Arquitectura de entorno. El marco de integración es un

conjunto de programas especializados que permiten a cada herramienta CASE comunicarse con las demás, para crear una base de datos de proyectos y mostrar una apariencia homogénea al usuario final (el ingeniero de software). Los servicios de portabilidad permiten que las herramientas CASE y su marco de integración puedan migrar a través de diferentes plataformas hardware y sistemas operativos sin grandes esfuerzos de adaptación. El sistema operativo es el software primario en donde se corre la aplicación o la herramienta, la plataforma de hardware hace alusión a todo el equipo físico de equipo de computo y periféricos necesarios para pueda funcionar adecuadamente el software instalado. Por último la arquitectura de entorno establece la relación de nuestro hardware con otras plataformas tomando en cuenta diferentes tipos de ambientes con el objetivo de mantener un flujo estable para que el la integración y portabilidad se mantenga en forma efectiva.

Básicamente la diferencia entre plataforma de hardware y la arquitectura de entorno es en cuanto a sus alcances, mientras una se dedica a trabajar en forma particular la otra prevé un circulo de acción a nivel general al entorno en que finalmente la herramienta de tipo CASE se va a desenvolver.

Los bloques de los que hemos hecho mención representan una base para la integración de las herramientas CASE. Sin embargo, la mayoría no han sido construidas utilizando todos los bloques componentes comentados anteriormente. De hecho, muchas de estas son soluciones "puntuales" i esto es, una herramienta se utiliza como ayuda en una actividad concreta de ingeniería del software, pero no se comunica directamente con otras herramientas i no está unida a una base de datos de proyectos y no parte de un entorno CASE integrado.

Aunque esta situación no es la ideal, una herramienta CASE puede ser utilizada eficientemente, aún siendo una solución puntual.

♦ CLASIFICACIÓN DE LAS HERRAMIENTAS CASE

Las herramientas CASE se pueden clasificar por su función, por su papel como instrumentos para el personal técnico o los directivos, por la arquitectura de entorno (hardware y software) que la soportan o incluso por su origen y costo. Para efectos explicativos se considerará la funcionalidad como criterio temporal haciendo hincapié en profundizar en las herramientas que consideramos apropiadas para el desarrollo del presente trabajo.

Herramientas de planificación de sistemas de gestión

Mediante la modelización de los requisitos de información estratégica de una organización, las herramientas de planificación de sistemas de gestión proporcionan un "metamodelo" del cual se pueden obtener sistemas de información específicos. En lugar de centrarse en los requisitos de una aplicación especifica, la información de gestión se modeliza según va pasando a través de distintas entidades organizativas de una compañía.

El objetivo principal de las herramientas de esta categoría son ayudar a comprender mejor cómo se mueve la información entre distintas unidades organizativas.

Herramientas de gestión de proyectos

Hoy en día, la mayoría de las herramientas CASE de gestión de proyectos se centran en un elemento específico de la gestión de proyecto, en lugar de proporcionar un soporte global para la actividad en gestión. Utilizando un conjunto seleccionado de herramientas CASE, el director de proyecto puede hacer estimaciones útiles de esfuerzo, costo, y duración de un proyecto,definir una estructura de partición de trabajo y hacer una planificación realista del mismo y hacer el seguimiento del proyecto de forma continua.

Además el director puede utilizar estas herramientas para recoger datos que le permitan realizar una estimación de la productividad del desarrollo y de la calidad del producto.
Con este tipo de herramientas CASE se puede hacer un seguimiento que va desde los requisitos de la petición de propuesta inicial del cliente, hasta el trabajo de desarrollo que convierte estos requisitos en producto final.

Herramientas de soporte

La categoría de herramientas de soporte engloba las herramientas de aplicación y de sistemas que complementan el proceso de ingeniería del software. Las herramientas que caen en esta amplia categoría recogen las actividades aplicables en todo el proceso de ingeniería de software.

Estas incluyen herramientas de documentación, herramientas de gestión de redes y software de sistema. herramientas de control de calidad y herramientas de gestión de base de datos y de configuración del software.

Herramientas de análisis y diseño

Las herramientas de análisis y diseño permiten al ingeniero de software crear un modelo de sistema que se va a construir. El modelo contiene una representación de los datos y el flujo de control, del contenido de los datos (a través de una definición de un diccionario de requisitos), representaciones de los procesos, especificaciones de control y otras representaciones del modelo. Las Herramientas de análisis y diseño permiten la creación de un modelo y también la evaluación de la calidad del modelo. Mediante la comprobación de la validez y la consistencia del modelo, estas herramientas, proporcionan al ingeniero cierto grado de confianza en la representación del análisis y ayudan a eliminar errores antes de que se propaguen al diseño, o lo que es peor, al código mismo.

En cuanto a las herramientas de diseño y desarrollo de interfaces son , en realidad, un conjunto de componentes de software, tales como menús, botones, estructuras de ventanas, iconos, mecanismos de visualización, controladores de dispositivos y otros elementos de este tipo. Sin embargo, estos conjuntos de herramientas están siendo reemplazados por herramientas para desarrollar prototipos que permitan la creación rápida en pantallas de interfaces sofisticadas ajustadas al estándar elegido por el software (por ejemplo : X - Windows).

Herramientas de programación

La categoría de herramientas de programación engloba los compiladores, los editores y los depuradores que se utilizan con los lenguajes de programación convencionales. Además, también entran en esta categoría los entornos de programación orientada a objetos (O.O), los lenguajes de cuarta generación, los generadores de aplicaciones y los lenguajes de consulta de base de datos.

Dentro de las herramientas de codificación de cuarta generación la tendencia hacia la representación de aplicaciones de software en niveles más altos de abstracción ha hecho que muchos diseñadores utilicen este tipo de herramientas.

Los sistemas de consulta a bases de datos, los generadores de código y los lenguajes de cuarta generación han cambiado la forma en que se desarrollan los sistemas. No hay duda de que el objeto final del CASE es la generación automática de código -- esto es la representación de sistemas a nivel de abstracción más alto que el de los lenguajes de programación convencionales. Idealmente, estas herramientas de generación de código no sólo traducirán la descripción de un sistema a un programa operativo, sino que también ayudarán a verificar la corrección de la especificación del sistema, de tal forma que la salida resultante satisfaga los requisitos del usuario.

Con el paso del tiempo, se prevé que con el uso de estas Herramientas cada menos código será escrito manualmente.

Por otro lado la programación orientada a objetos se viene considerando como una de las tecnologías más actuales de la ingeniería de software ; por esta razón, los vendedores del sistema CASE están lanzando al mercado nuevas herramientas para el desarrollo de software orientado a objetos.

Los entornos de programación orientada a objetos suelen estar unidos a los lenguajes de programación específicos (por ejemplo C++). Un entorno O.O. típico incorpora características de los Interfaces de tercera generación (ratón, ventanas, menús desplegables, operaciones sensibles al contexto, multitarea, etc.) con funciones especializadas como la de "inspector", una función que permite al ingeniero de software examinar todos los objetos contenidos en unas bibliotecas de objetos para determinar si pueden ser reutilizados en la aplicación actual.

Herramientas de integración y prueba.

En su directorio de herramientas de prueba de software, el libro "Software quality Engineering" [SQE90] define las siguientes categorías :

Adquisiciones de datos: Herramientas que adquieren datos para ser usados durante la prueba.

Medida estática :Herramientas que analizan el código fuente sin ejecutar casos de prueba.

Medida dinámica:Herramientas que analizan el código fuente durante la ejecución.

Simulación: Herramientas que simulan la función del hardware o de otros elementos externos.

Gestión de pruebas: Herramientas que ayudan a la planificación, el desarrollo y el control de las pruebas.

Herramientas de funcionalidad: Herramientas que realizan varias de las funciones anteriores.

Herramientas de creación de prototipos

La realización de prototipos es un paradigma de la ingeniería de software ampliamente utilizado, y como tal, cualquier herramienta enfocada a ello puede ser denominada legítimamente, una herramienta de creación de prototipos ; lo que si podemos asegurar es que según evolucionan este tipo de herramientas, parece razonable pensar que algunas de ellas serán específicas de un campo. Esto es, la herramienta estará diseñada para ser aplicada en una área restringida de aplicación.

Para los próximos años será frecuente encontrar herramientas específicas para el área de las telecomunicaciones, aplicaciones espaciales, automatización de factorías y otras muchas. Estas herramientas utilizarán una base de conocimiento que "entienda" el campo de aplicación, facilitando la creación de prototipos.

Herramientas de mantenimiento

Las herramientas CASE para el mantenimiento de software abarcan una actividad que actualmente ocupa, aproximadamente, el 70 % del esfuerzo total dedicado al software. La categoría de herramienta de mantenimiento puede subdividirse de la siguiera forma :

Herramienta de ingeniería inversa a especificaciones: Toman el código fuente como entrada y generan modelos de diseño y análisis estructurado, listas de utilización y otra información relacionada con el diseño.

Herramientas de reestructuración y análisis de código : Analizan la sintaxis del programa, generan un grafo de flujo de control y un programa estructurado.

Herramientas interactivas de reingeniería del sistema: Se utilizan para modificar sistemas de bases de datos (conversión de archivos).

Estas herramientas están limitadas a lenguajes de programación específicos (aunque incluyen los lenguajes más utilizados) y requieren cierto grado de interacción con el ingeniero de software.

La siguiente generación de herramientas de ingeniería inversa y de reingeniería harán un uso más intenso de las técnicas de inteligencia artificial, aplicando uina base de conocimiento específica del campo.

Herramientas de estructura

Las herramientas de esta categoría tienen componentes funcionales para el tratamiento de datos, de interfaces y con capacidad de integración con otras herramientas. La mayoría trabajan con bases de datos orientadas a objetos y con un conjunto interno de herramientas para establecer interfaces con herramientas de otros suministradores de sistemas CASE ; la mayoría tiene capacidad de gestionar la configuración, permitiendo al usuario realizar cambios en los elementos de configuración de todas las herramientas CASE que se integren con la herramienta de estructura.

CASE E INTELIGENCIA ARTIFICIAL

.Algunas herramientas CASE incorporan sistemas expertos, pero la gran mayoría hace un uso muy reducido de las técnicas de inteligencia artificial.(IA) Muchas de las que si las hacen utilizan esta tecnología para la validación gráfica de modelos de diseño y análisis, aplicando reglas de diseño específicas de un método a los modelos creados por el ingeniero de software.

Los investigadores están evaluando entornos de programación que hacen uso de "agentes" de diseño y evaluación (herramientas inteligentes que ayudan en el análisis, el diseño y la prueba de sistemas basados en computadora). En lugar de evaluar simplemente el modelo que creó una persona, un agente ayudará al ingeniero en su labor de resolución de problemas.

Otras áreas de aplicación para el software de IA es el reconocimiento de patrones (imágenes y voz), la prueba de teoremas y los los juegos. En los últimos años ha surgido una nueva rama del software de IA llamada *redes neuronales artificiales* [WAS89]. Una red neuronal simula la estructura de proceso del cerebro (las funciones de la neurona biológica) y a la larga puede llevar a una clase de software que pueda reconocer patrones complejos y aprender de "experiencia" pasada

CONCLUSIONES

El software de desarrollo implica un conjunto de herramientas que traducen, detectan errores, optimizan, etc., los programas escritos por el usuario al lenguaje de las computadoras; por ejemplo los compiladores e interpretes traducen programas, hechos en lenguaje de alto nivel, al lenguaje que entiende la máquina (lenguaje máquina); los ensambladores traducen el lenguaje ensamblador a lenguaje máquina, crean un programa que se puede ejecutar en la computadora, y lo cargan en su memoria ; los depuradores y trazadores ayudan a la detección de errores en los programas y al seguimiento de la ejecución de éstos.

La ingeniería del software es una disciplina que integra métodos, herramientas y procedimientos para el desarrollo de software de computadora.

Se han propuesto varios modelos para el desarrollo de software distintos, cada uno con sus propias ventajas y desventajas, pero todos tienen una serie de fases genéricas en común, y es importante que se lleven a cabo y se les dé la importancia que merecen cada una de ellas para garantizar que el desarrollo del software se esté haciendo conscientemente, con responsabilidad, con un nivel básico de control, y sobre todo con una gran inclinación hacia la calidad.

El hacer referencia a la calidad no resulta un lujo sino una necesidad en los tiempos actuales, el hacer las cosas pensando en que deben de funcionar de manera adecuada es imperativo para las empresas evitando a toda costa el estar poniendo parches en los sistemas.

Muy probablemente aunque en sus inicios un sistema se hubiera conceptualizado como bueno y al final fuera un fracaso, pudo deberse en gran medida a que el problema se derivó en no llevar un seguimiento que midiera la calidad con que se desenvolviera los procesos, redundando en un software inestable.

Ha sido interesante haber analizado el impacto en el costo derivado del mantenimiento del software defectuoso, así como las posibles medidas para su posible corrección, las revisiones que éstas deban tener y las directrices que marquen la pauta de como hacer frente ante estas circunstancias.

El hecho de contar con métricas de calidad nos permitieron ver el impacto sobre los posibles errores que pudieran aparecer, es así la necesidad de contar con procesos y alternativas vistas en los índices de calidad y que con ayuda de muestreos estadísticos , nos alerten cuando ocurra algo fuera de lo normal.

La fiabilidad del software va encaminada hacia el mantenimiento de una confianza hacia los programas , tomando como referencia los puntos anteriores. Realmente en la medida que nosotros hagamos caso de estos conocimientos vamos a ver que el software no será un enemigo con el que estemos batallando, sino que se puede convertir en un aliado bastante poderoso, pero para que esto sea una realidad debemos de contar con ciertas medidas y modelos de fiabilidad que nos ayuden a hacer del software algo seguro.

Aunque el uso de la tecnología CASE todavía no es una realidad en todas las empresas, si podemos pensar que en un futuro cercano resultarán indispensables como auxiliares en la elaboración del diseño, en el desarrollo, y en el mantenimiento de los sistemas.

Por último cabe señalar que el sólo hecho de cumplir con los pasos de las técnicas de ingeniería del software y con el modelo de aseguramiento de la calidad basado en ISO 9000 no nos asegura como resultado un producto de alta calidad, sin embargo nos proporciona un buen comienzo en la búsqueda de la calidad y nos garantiza que se cumplen los requisitos mínimos para alcanzar un resultado satisfactorio.

B I B L I O G R A F I A

* Computer-Aided Software Design: Build Quality Software with CASE.
  Schindler Max.
  Ediciones J. WILEY

* Ingeniería del Software e Inteligencia Artificial.
  CETTICO (Centro de Transferencia Tecnológica en Informática
  y comunicaciones).
  Ed. CULTURAL, S.A.

* An ISO 9000 Approach to Building Quality Software.
  Oesten Oskarsson/Robert L. Glass.
  Ed. Prentice Hall PTR.

* Ingeniería del Software Práctico y conciso.
  Mahnke Hans
  Ed DATANET

* Herramientas CASE
  Williams S. Davis
  Ed. PARANINFO

REFERENCIAS

[MAN84] M. Walston C. "A Method for Programming Measurement and Estimation". IBM System Journal.

[TAJ84] Tajima D.T. "The Computer Software Industry in Japan". 1984

[Eva87] Evans, W. Vesely. "Software Engineering". Addison Wesley 1987

[BRO75] Brosili V. And M. Zelkowitz. "Analizing Medium Scale Software Development". IEE,1975

[MCC77] McCall J. "Factors in Software Quality". NTIS AD-AO49, 1977

[GRA87] Grady, R.B. "Software Metrics: Establishing a Company-Wide"

Prentice-Hall, 1987

[IEE89] "Software Engineering Standars". IEE,1989

[CAV78] J. O. Cavano and McCall. "A Framework for the Measurement of Software Quality"., 1978

[MUS87] Musa J. D. "Engineering and Managing Software with Reliability Measures"., McGraw-Hill, 1987

[LEV86] Leveson N. G. "Software Safety: Why, What and How"., 1986.