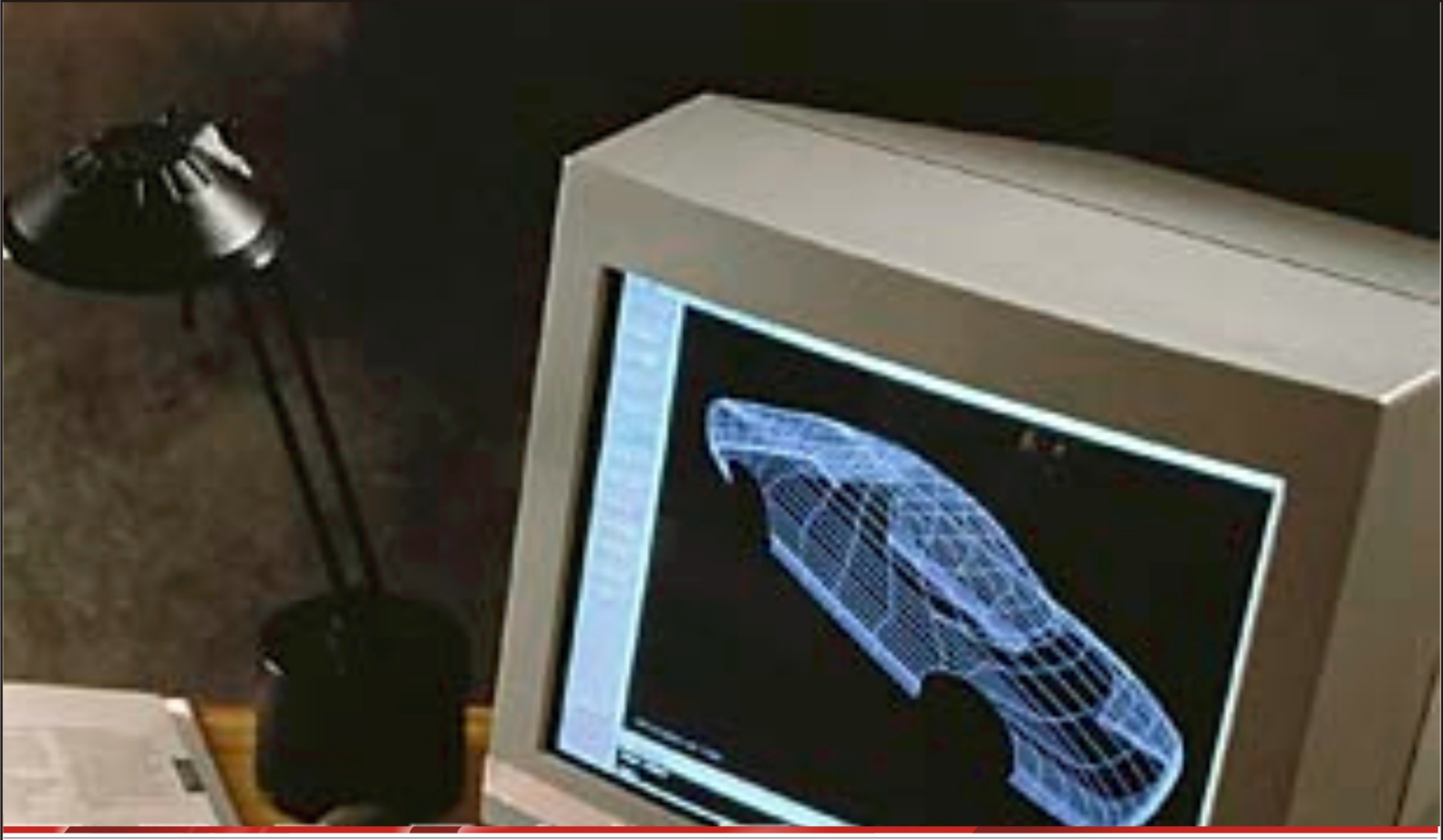


INGENIERÍA DE SOFTWARE

TINS

UNIVERSIDAD TECNOLOGICA DEL PERU

UNIVERSIDAD TECNOLÓGICA DEL PERU

Vicerrectorado de Investigación

INGENIERÍA DE SOFTWARE

TINS Básicos

INGENIERÍA DE SISTEMAS

TEXTOS DE INSTRUCCIÓN BÁSICOS (TINS) / UTP

Lima - Perú

© INGENIERÍA DE SOFTWARE

Desarrollo y Edición: Vicerrectorado de Investigación

Elaboración del TINS: Profesor Hernán Robalino Gómez

Diseño y Diagramación: Julia Saldaña Balandra

Soporte académico: Instituto de Investigación

Producción: Imprenta Grupo IDAT

PRESENTACIÓN

El presente texto elaborado en el marco de desarrollo de la Ingeniería, es un material de ayuda instruccional, en la carrera de Ingeniería de Sistemas para la Asignatura de Ingeniería de Software, en el sétimo ciclo de estudios.

Plasma la iniciativa institucional de innovación de la enseñanza-aprendizaje educativa universitaria, que en acelerada continuidad promueve la producción de materiales educativos, actualizados en concordancia a las exigencias de estos tiempos.

Esta primera edición apropiadamente recopilada, de diversas fuentes bibliográficas, de uso frecuente en la enseñanza de la Ingeniería de Sistemas, está ordenada en función del syllabus de la Asignatura, arriba mencionada.

La conformación del texto ha sido posible gracias al esfuerzo y dedicación académica del profesor Hernán Robalino Gómez; contiene siete capítulos, cuyas descripciones genéricas son como sigue:

El CAPÍTULO I es un panorama breve de la ingeniería de software, incluyendo su evolución, crisis y futuro. Además las etapas y procesos de la ingeniería de software.

En el CAPÍTULO II se tiene un resumen de las metodologías para el desarrollo de software, considerando algunos modelos del proceso de desarrollo de software.

En el CAPÍTULO III se hace una revisión de los principales procesos de la ingeniería de software, así como la gestión del proyecto de software incorporando la programación de proyectos con PERT – CPM.

El CAPÍTULO IV presenta los conceptos del análisis de requisitos que son los que orientan el desarrollo del software. Se incluye los conocimientos del análisis estructurado y del análisis orientado a objetos.

En el CAPÍTULO V se presenta dos formas de diseñar software, es decir: el diseño estructurado y el diseño orientado a objetos.

El CAPÍTULO VI es una revisión de los conceptos de base de datos, incluyendo el proceso de normalización. También se presentan diferentes criterios para el diseño de interfaces de usuario, así como la usabilidad del software.

En el CAPÍTULO VII se desarrolla diversos criterios para medir la calidad del software. Se establecen diferentes tipos de pruebas del software y del sistema. Se incluyen la forma de llevar a cabo la configuración y mantenimiento del software. Por ultimo se desarrolla y presentan diversas herramientas CASE que apoyan el desarrollo de software.

Finalmente, al cierre de estas líneas, el agradecimiento Institucional, por la labor cumplida al profesor Ing. Hernán Robalino Gómez; así mismo el reconocimiento a quienes han hecho posible la presente edición.

Vicerrectorado de Investigación

ÍNDICE

DISTRIBUCIÓN TEMÁTICA

Clase N°	Tema	Semana	Horas
1	Introducción a la Ingeniería de Software: Conceptos generales sobre programa, software e Ingeniería de software. Etapas y procesos de la ingeniería de software. Paradigmas de la ingeniería de software. Enfoques para el desarrollo de software.	1	02
2	Metodologías para el desarrollo de Software. Análisis y diseño estructurado. Proceso Unificado Rational (RUP). Métrica 3. Proyecto de Desarrollo de Software. Estudio de factibilidad. Factibilidad Técnica. Factibilidad Económica. Factibilidad Operativa.	2	02
3	Revisión de los Principales procesos de la Ingeniería de Software: - Análisis de requisitos - Análisis y diseño del Software - Pruebas del software - Calidad del software - Configuración - Documentación del software - Mantenimiento del software	3	02
4	La gestión del proyecto de software. Diagramas PERT. 1.-Planificación de actividades 2.-Estimación de tiempos y recursos 3.-Métricas	4	02
5	Análisis de requisitos 1.- Modelado del negocio. Técnica IDEF0 y BPMN 2.- Fundamento del análisis de requisitos. 3.- Requerimientos Funcionales y No Funcionales. ANÁLISIS DEL SISTEMA: 1.-Los pasos del análisis del sistema 2.-Modelación de la arquitectura del sistema 3.-Especificación del sistema.	5	02
6	ANÁLISIS ESTRUCTURADO. Diagrama de Flujo de Datos (DFD), Diagr. Entidad-Relación (ERD) y diccionario de datos. ANÁLISIS ORIENTADO A OBJETOS. Diagramas UML. Diagrama Casos de Uso - Diagr. Secuencia - Diagr. Actividades - Diagr. Clases.	6	02

Clase N°	Tema	Semana	Horas
7	DISEÑO ESTRUCTURADO. 1.-Fundamento del diseño de software 2.-Diseño orientado al flujo de datos. DISEÑO ORIENTADO A OBJETOS. 1.-Diagrama de clases. 2. Especificación de operaciones. 3.-Diagrama de Componentes. 4. Diagrama de Despliegue.	7	02
8	Revisión de los conceptos de Base de Datos, Normalización y desnormalización de una BD. Formas normales: 1FN, 2FN, 3FN, 4FN y 5FN. Desnormalización hacia arriba, hacia abajo, intratabla, divide y vencerás.	8	02
9	Diseño de interfaces de usuario (GUI's). Estándares para el diseño de GUI's y usabilidad.	9	02
10	**EXAMEN PARCIAL**	10	**02**
11	Usabilidad del software.	11	02
12	La calidad del software: - El modelo de madurez de capacidades (CMM) - Estándares para la calidad del software - La norma técnica peruana Nº 12207	12	02
13	Métricas del Software	13	02
14	Las pruebas del software - Tipos de pruebas - El proceso de pruebas: Estrategia, planeación y construcción de la prueba.	14	02
15	Configuración y mantenimiento del software	15	02
16	Documentación del Software	16	02
17	Herramientas CASE para el desarrollo de software	17	02
18	Integración de Trabajo en base a los documentos del proyecto	18	02
19	**EXAMEN FINAL**	19	**02**

CAPÍTULO I

1.1 INTRODUCCIÓN

Los negocios enfrentan una paradoja, tienen oportunidades nunca vistas para aprovechar los nuevos mercados, y entre tanto, los mercados tradicionales cambian de manera sustancial, reduciéndose o haciéndose intensamente competitivos. Además, los reducidos márgenes de beneficios paralelos a las crecientes exigencias del cliente por productos y servicios de calidad, determinan presiones inexorables en muchas empresas.

Las empresas ya no limitan su crecimiento a las bases tradicionales del cliente. Los banqueros ofrecen servicios de corretaje y seguros. Las compañías de seguros mercadean servicios financieros. Las compañías de alta tecnología venden bienes de consumo a los clientes. La competencia puede surgir de manera inesperada en cualquier lugar. Esto significa que las empresas ya no pueden confiarse demasiado con respecto a sus participaciones de mercado y a sus posiciones competitivas.

La apertura de los mercados mundiales ha hecho tambalear muchas corporaciones, generando virtualmente una masiva reestructuración en cada sector de negocios. Varias empresas están reestructurando sus costos básicos mediante una severa disminución de tamaño.

La vieja empresa está pobremente equipada para responder a las nuevas necesidades de los negocios. La organización jerárquica clasifica a las personas en dos grupos: los gobernantes y los gobernados. Al final de la cadena de comando se encuentra el gobernante supremo; en el otro extremo se hallan los absolutamente gobernados. En medio de esta cadena están las personas que actúan en forma alternada como dirigentes o gobernados. Estos son los gerentes de los niveles medios que actúan como transmisores de las comunicaciones que vienen desde la cúpula.

La comunicación en el otro sentido se encuentra limitada, excepto por las relaciones formales de trabajo y administración. Aunque esta imagen puede parecer estereotipada, éste era el modelo tradicional de la empresa. Hoy día, existe una creciente aceptación de que esta estructura acaba la creatividad, la automotivación, el compromiso y la responsabilidad hacia las exigencias del mercado, para no mencionar la falla de suplir las necesidades humanas para completar el trabajo. La estructura de una nueva empresa se desplaza de una jerarquía multilateral a negocios con estructuras planas interconectadas, relativamente autónomas. El concepto de la organización se amplía para incluir vínculos con socios externos de los negocios: proveedores y cliente.

El punto central de los recursos se desplaza del capital a los recursos humanos y de información. El profesional emerge como el protagonista central; a menudo trabaja en equipo multidisciplinarios que trascienden las barreras organizacionales tradicionales. Los individuos son habilitados y motivados para actuar, y lo hacen de manera responsable y creativa. Libres del control burocrático, ellos toman la iniciativa e incluso asumen los riesgos para estar más cerca de los clientes y trabajar con mayor

productividad. Ellos se motivan entre sí para alcanzar objetivos de grupo en vez de satisfacer a los superiores. Con intereses comunes que sean inmediatos y claros, prospera la cooperación.
La nueva estructura es posible cuando cada miembro comprende la visión de equipo, cuenta con la competencia exigida, tiene la confianza de los demás y, algo muy importante, tiene acceso a la información y las herramientas que requieren para funcionar y colaborar ampliamente dentro del equipo.

Mediante cuál tecnología y cuáles medios ocurrirá el cambio hacia la nueva empresa? es evidente que está presentándose un nuevo paradigma en la situación geopolítica del mundo. El ascenso de la nueva empresa abierta e interconectada constituye un nuevo paradigma organizacional.

También a ocurrido un cambio en cuanto a quién utiliza los computadores. Los usuarios desean modelar la tecnología que se implementa en sus organizaciones, desean controlar su uso y determinar el efecto que tendrá en su propio trabajo, rápidamente comprenden que su uso efectivo de la tecnología acoplada a un cambio en la manera como ellos desempeñan sus negocios determinará su éxito personal y organizacional.

Antes los sistemas se consideraban como algo interno para la organización, como reflejo de las barreras existentes de las empresas. Ahora, los sistemas se amplían a las organizaciones exteriores para vincular las empresas con sus proveedores, canales de distribución y clientes. Las compañías de seguros y las aerolíneas se encuentran vinculadas con agentes. Los bancos suministran acceso en línea a los clientes.

La investigación halló que tales sistemas pueden fortalecer la lealtad del cliente; bloquear a los competidores, acelerar la distribución de bienes y servicios al cliente y ahorrar dinero.

La tecnología se está convirtiendo en un vehículo para generar entre los socios comerciales, tanto proveedores como clientes de productos y servicios.

Las tecnologías emergentes incluyen bases de datos interempresariales, sistemas de respuesta oral, mensajes electrónicos y nuevas tecnologías en el punto de venta. Estándares como el intercambio electrónico de datos (EDI) - el intercambio de documentos comerciales de computador a computador entre las compañías - transforman las maneras como funcionan las compañías entre sí. Por ejemplo, cuando los grandes productores de automóviles solicitan que sus proveedores se comuniquen con ellos utilizando el EDI, uno de los objetivos consiste en hacer que los proveedores sean más productivos, obtengan más ganancias y, por consiguiente, tengan mayor estabilidad. Los fabricantes de automóviles se interesan en la productividad de sus proveedores y pueden contribuir a ello mediante el EDI. De este modo en este momento se generan nuevas empresas ampliadas.

En general, la tecnología de la información puede considerarse como clases de sistemas que van desde el nivel personal hasta los sistemas interempresariales. Las aplicaciones personales directamente apoyan y son controladas por el usuario final. Las aplicaciones del trabajo de grupo se comparten en equipo o funciones que pueden localizarse de manera centralizada o extenderse ampliamente en toda la empresa. Las aplicaciones corporativas o empresariales apoyan un amplio rango de usuarios en toda la empresa y pueden incluir muchas divisiones o departamentos. Las aplicaciones

públicas o interempresariales involucran interacción con los usuarios y con sistemas externos a la organización

1.2 INTRODUCCIÓN A LA INGENIERÍA DE SOFTWARE

Según la definición del IEEE, "software es la suma total de los programas de ordenador, procedimientos, reglas, la documentación asociada y los datos que pertenecen a un sistema de cómputo" y "un producto de software es un producto diseñado para un usuario". En este contexto, la Ingeniería de Software (SE del inglés "Software Engineering") es un enfoque sistemático del desarrollo, operación, mantenimiento y retiro del software.

DEFINICIONES DE BOEHM

- ***Software*** *es el conjunto de programas, procedimientos y documentación asociados a un sistema, y particularmente a un sistema computacional.*

- ***Ingeniería*** *es la aplicación de la ciencia y las matemáticas mediante lo cual las propiedades de la materia y las fuentes de energía de la naturaleza se hacen útiles al hombre en estructuras, máquinas, productos, sistemas y procesos.*

- ***Ingeniería de software*** *es la aplicación de la ciencia y las matemáticas mediante la cual la capacidad de los equipos computacionales se hacen útiles al hombre a través de programas de computador, procedimientos y la documentación asociada.*

DEFINICION DE BAUER
Ingeniería del Software *es el establecimiento y uso de firmes principios y métodos de ingeniería para la obtención económica de software fiable y que funcione en máquinas reales*.

Durante los primeros años de la informática, el software se consideraba como un añadido. La programación era un "arte", para el que no existían metodologías, era un proceso que se realizaba sin ninguna planificación. En esta época toda la programación se desarrollaba a medida para cada aplicación, y en consecuencia tenía muy poca difusión, habitualmente quien lo escribía era porque lo necesitaba, y era quien lo mantenía.

La ingeniería de software se debe a que el entorno actual de desarrollo de sistemas software viene adoleciendo de:

- Retrasos considerables en la planificación
- Poca productividad
- Elevadas cargas de mantenimiento
- Demandas cada vez más desfasadas con las ofertas
- Baja calidad y fiabilidad del producto
- Dependencia de los realizadores

Esto es lo que se ha denominado comúnmente "Crisis del Software",

1.2.1 La evolución del software

El contexto en que se ha desarrollado el software está fuertemente ligado a las casi cinco décadas de evolución de los sistemas informáticos. Un mejor rendimiento del hardware, una reducción del tamaño y un coste más bajo, han dado lugar a sistemas informáticos más sofisticados.

A continuación describiremos la evolución del Software dentro del contexto de las áreas de aplicación de los sistemas basados en computadoras.

Los primeros años (1950 - 1965):

- Sistemas orientados por lotes con excepción del sistema de reservaciones de billetes de la American Airlines y los sistemas de tiempo real para la defensa (SAGE).
- Hardware dedicado a la ejecución de una aplicación específica.
- Distribución limitada al entorno de la aplicación.
- Software "a medida" para cada aplicación.

La segunda era (1965 - 1975):

- Multiprogramación y sistemas multiusuarios introducen nuevos conceptos de interacción hombre-máquina.
- Sistemas de tiempo real que podían recoger, analizar y transformar datos de múltiples fuentes.
- Avances en los dispositivos de almacenamiento en línea condujeron a la primera generación de sistemas de gestión de Base de Datos.
- Software como producto y la llegada de las "casas de software" produciéndose así una amplia distribución en el mercado.

Conforme crecía el número de sistemas informáticos, comenzaron a extenderse las bibliotecas de software de computadora. Las casas desarrollaban proyectos en que se producían programas de decenas de miles de sentencias fuente. Los productos de software comprados en el exterior incorporaban cientos de miles de nuevas sentencias. Una nube negra apareció en el horizonte. Todos estos programas tenían que ser corregidos cuando se detectaban fallos, modificados cuando cambiaban los requisitos de los usuarios o adaptados a nuevos dispositivos de hardware que se hubiera adquirido. Estas actividades se llamaron colectivamente *mantenimiento del software.*

La tercera era (1975 - 1985):

- Procesamiento Distribuido. Múltiple computadoras, cada una ejecutando funciones concurrentes y comunicándose con alguna otra.
- Redes de área local y de área global. Comunicaciones digitales de alto ancho de banda y la creciente demanda de acceso "instantáneo" a los datos.
- Amplio uso de microprocesadores y computadoras personales (hardware de bajo costo). Incorporación de "inteligencia" (autos, hornos de microondas, robots industriales y equipos de diagnóstico de suero sanguíneo). Impacto en el consumo.

La cuarta era (1985 -2000):

- Tecnología orientada a objetos
- Los sistemas expertos y la inteligencia artificial se han trasladado del laboratorio a las aplicaciones prácticas.
- Software para redes neuronales artificiales (simulación de procesamiento de información al estilo de como lo hacen los humanos).

1.2.2 Aplicaciones del software

Para determinar la naturaleza de una aplicación de software, hay dos factores importantes que se deben considerar: el contenido y el determinístico de la información. El *contenido* se refiere al significado y a la forma de la información de entrada y salida. El *determinístico de la información* se refiere a la predecibilidad del orden y del tiempo de llegada de los datos.

Las siguientes áreas del software indican una amplitud de las posibilidades de aplicación.

- Software de sistemas. Es un conjunto de programas que han sido escritos para servir a otros programas. Por ejemplo: compiladores, editores, etc.
- Software de tiempo real. Es el software que mide/analiza/controla sucesos del mundo real conforme ocurren.
- Software de gestión. Constituye la mayor área de aplicación del software. Los sistemas "discretos" (ejemplo: nóminas, cuentas de haberes/débitos, inventarios, etc.) han evolucionado hacia el software de sistemas de información de gestión (SIG), que acceden a una o más bases de datos grande que contienen información comercial.
- Software de ingeniería y científico. Está caracterizado por los algoritmos de "manejo de números". Abarca aplicaciones a todas las ciencias hasta el diseño asistido por computadora (CAD), la simulación de sistemas y otras aplicaciones interactivas.
- Software empotrado. Reside en la memoria de sólo lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo.
- Software de computadoras personales. El procesamiento de textos, las hojas de cálculo, los gráficos por computadoras, entretenimiento, gestión de BD, aplicaciones financieras, de negocios y personales, redes, etc.
- Software de inteligencia artificial. Hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. Actualmente el área más activa es la de los *sistemas expertos o sistemas basados en el conocimiento*.

1.2.3 Crisis del Software

La crisis del software 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.

Los problemas que afligen el desarrollo del software se pueden caracterizar bajo muchas perspectivas, pero los responsables de los desarrollos de software se centran sobre los aspectos de "fondo":

- la planificación y estimación de los costes son frecuentemente imprecisas;
- la "productividad" de la comunidad del software no se corresponde a la demanda de sus servicios, y
- la calidad del software no llega a ser a veces ni aceptable.

Tales problemas son sólo las manifestaciones más visibles de otras dificultades del software:

- No tenemos tiempo para recoger los datos sobre el proceso de desarrollo del software.
- La insatisfacción del cliente del sistema "terminado" se produce demasiado frecuente.
- La calidad del software es normalmente cuestionable.
- El software existente puede ser muy difícil de mantener.

Los problemas asociados con la crisis del software se han producido por el propio carácter del software y por los errores de las personas encargadas del desarrollo del mismo.

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. 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 resultado una pobre calidad y mantenibilidad del software.

1.2.4 Mitos del Software

Los mitos del software tienen varios atributos que los hacen peligrosos. tuvieron un sentido intuitivo, hoy la mayoría de los profesionales reconocen a los mitos por lo que son actitudes erróneas que han causado serios problemas tanto a los gestores como a los técnicos

Mitos de gestión

- Por qué debemos cambiar nuestra forma de desarrollar el software?
  Estamos haciendo el mismo tipo de programación ahora que hace diez años

- Tenemos ya un libro que está lleno de estándares y procedimientos para construir software
- Nuestra gente dispone de las herramientas de desarrollo de software más avanzadas
- Si fallamos en la planificación, podemos añadir más programadores y adelantar el tiempo perdido

Mitos del cliente

- Una declaración general de los objetivos es suficiente para comenzar a escribir los programas
- Los requerimientos del proyecto cambian continuamente

Mitos de los realizadores

- No hay realmente ningún método para el análisis, diseño y prueba de función
- Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado
- Hasta que no tenga el programa ejecutándose realmente no tengo forma de establecer su calidad
- Lo único que se entrega al terminar el proyecto, es el programa funcionando
- Una vez que el software se esta usando, el mantenimiento es mínimo

1.3 PARADIGMA DE LA INGENIERÍA 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.

La ingeniería del software surge de la ingeniería de sistemas y de hardware. Abarca un conjunto de tres elementos claves: 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.

Los Métodos.- De la IS indican "como" 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 programa y procedimientos algorítmicos, codificación, prueba y mantenimiento. Los métodos de la IS introducen frecuentemente una notación especial orientada a un lenguaje o gráfica y un conjunto de criterios para la calidad de software.

Las Herramientas.- De la IS suministran un soporte automático semiautomático para los métodos. Hoy existen herramientas para soportar cada uno de los elementos mencionados anteriormente. Cuando se integran las herramientas de forma que la información creada por herramientas pueda ser usada por otra, se establece un sistema para el soporte del desarrollo del software, llamado ingeniería de software asistida por computadora (CASE). CASE combina software, hardware y BD.

Los Procedimientos.- De la IS 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 ayuden a los gestores del software a evaluar el progreso.

1.3.1 Futuro de la ingeniería de software

El obstáculo más inmediato del futuro en la ingeniería del software será la reducción del tiempo de mercado de las soluciones de software. Habrá necesidad de estar dentro de una velocidad creciente no solo de desarrollo sino también de innovación.

Dentro de 10 años la sociedad de ese tiempo será totalmente dependiente de una infraestructura controlada por los sistemas de software. Será imposible regresar a los sistemas manuales. La implicación de estas aseveraciones es que la disponibilidad, confiabilidad y seguridad de esos sistemas tendrá que ser muy alto.

Si la industria del software va a encontrar el obstáculo de la nueva década tendrá que mejorar este hecho no solo al usar las mejores técnicas disponible e implementar nuevas tecnologías si no mejorar el proceso de desarrollo. El objeto de este mejoramiento será acelerar la obtención de los productos de software y asegurar que contribuirán a la satisfacción de los requerimientos percibidos del cliente.

La forma como controlamos la implementación de nuevas tecnologías de software y el proceso de software es por medio de la administración de proyecto.

Hay herramientas de administración de proyectos más que suficientes; algunos métodos y metodologías están disponibles para el ingeniero del software. El problema básico es que todas esas herramientas están basadas en sistemas que fueron utilizados originalmente en la ingeniería civil.

La administración de proyectos de software debe de ser considerada como una herramienta que recién esta desarrollándose y por lo tanto como una disciplina riesgosa.

No podemos administración en forma adecuada el software sin conocimientos de esos riesgos. Un método para controlar esos riesgos es la administración de proyectos basadas en el riesgo.

1.3.2 Administración de proyectos basadas en el riesgo

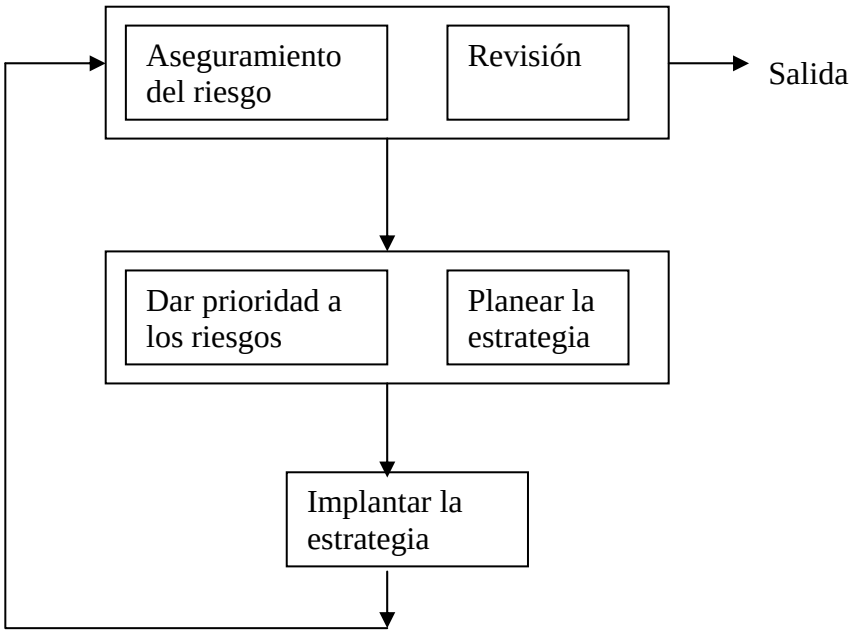
La administración del riesgo es una actividad de un ciclo de vida completo es una parte integral de un buen proyecto de administración, es el proceso de asesorar y controlar los proyectos de riesgo a fin de minimizar la oportunidad de desastre, los costos de volver a trabajar y horarios de tiempo extras y maximizar la oportunidad de completar el proyecto con calidad. El área donde el software innovador difiere por ejemplo:

La ingeniería civil es aquella que es altamente innovadora y aún es actual el viejo adagio de que “no se puede tener innovación sin riesgo por lo tanto el riesgo es endémico en la ingeniería del software”.

El proceso de la administración de proyecto basada en el riesgo puede ser dividido en dos fases principales cada una consistente de 3 pasos:

Enunciado del riesgo: Consistente en la identificación del riesgo, análisis y especificación de prioridades.
Control del riesgo: Consistente en la planeación de la administración del riesgo, respuesta al riesgo o resolución y monitoreo del riesgo.
El análisis del riesgo: consiste de la estimación, la probabilidad de ocurrencia y el efecto potencial del proyecto de cada riesgo.

Diagrama de Administración basada en el riesgo



Llegar hasta este proceso en la administración del proyecto basada en el riesgo tiene beneficios reales. Esto es, aumenta el conocimiento del personal del proyecto acerca de los riesgos potenciales.

La identificación del riesgo y el análisis no son pasos suficientes para permitir una efectiva administración de proyectos basada en el riesgo.

Los beneficios de la administración de proyectos basada en el riesgo se obtienen a través de la vida del proyecto en tanto los planes estén disponibles en avance para tratar con los riesgos.

1.4 ETAPAS Y PROCESOS DE LA INGENIERÍA DE SOFTWARE

El proceso de desarrollo del software contiene tres fases genéricas, independientemente del paradigma de ingeniería elegido. Las tres fases, *definición, desarrollo* y *mantenimiento*, se encuentran en todos los desarrollos del software, independientemente del área de aplicación, del tamaño del proyecto o de la complejidad.

1.- Pasos en la *definición* (se centra en el qué):
- Análisis del sistema
- Planificación del proyecto de software
- Análisis de los requisitos

2.- Pasos en el *desarrollo* (se centra en el cómo):
- Diseño del software
- Codificación
- Prueba del software

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

- Corrección
- Adaptación
- Mejora

CAPÍTULO II

2.1 METODOLOGÍAS PARA EL DESARROLLO DE SOFTWARE

Ciclo de Vida del Software.

Un modelo de ciclo de vida define el estado de las fases a través de las cuales se mueve un proyecto de desarrollo de software. El primer ciclo de vida del software, “Cascada”, fue definido por Winston Royce a fines del 70. Desde entonces muchos equipos de desarrollo han seguido este modelo. Sin embargo, ya desde 10 a 15 años atrás, el modelo cascada ha sido sujeto a numerosas críticas, debido a que es restrictivo y rígido, lo cual dificulta el desarrollo de proyectos de software. En su lugar, muchos modelos nuevos de ciclo de vida han sido propuestos, incluyendo modelos que pretenden desarrollar software más rápidamente, o más incrementalmente o de una forma más evolutiva, o precediendo el desarrollo a escala total con algún conjunto de prototipos rápidos.

Definición de un Modelo de Ciclo de Vida.

Un modelo de ciclo de vida de software es una visión de las actividades que ocurren durante el desarrollo de software, intenta determinar el orden de las etapas involucradas y los criterios de transición asociadas entre estas etapas. Un modelo de ciclo de vida del software:

- Describe las fases principales de desarrollo de software.
- Define las fases primarias esperadas de ser ejecutadas durante esas fases.
- Ayuda a administrar el progreso del desarrollo, y
- Provee un espacio de trabajo para la definición de un detallado proceso de desarrollo de software.

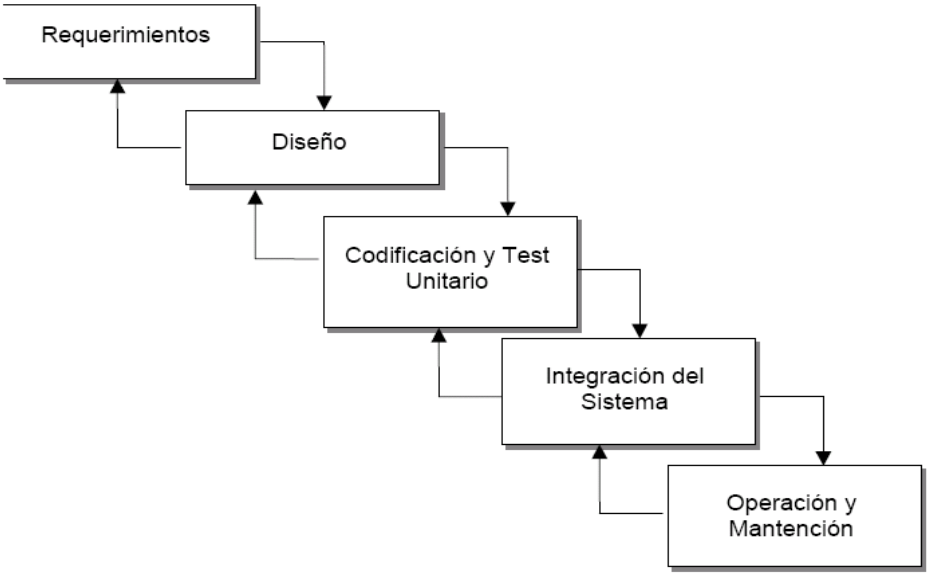
Así, los modelos por una parte suministran una guía para los ingenieros de software con el fin de ordenar las diversas actividades técnicas en el proyecto, por otra parte suministran un marco para la administración del desarrollo y el mantenimiento, en el sentido en que permiten estimar recursos, definir puntos de control intermedios, monitorear el avance, etc.

2.1.1 El ciclo de vida básico

Llamado también "modelo de cascada", este es el más básico de todos los modelos y sirve como bloque de construcción para los demás modelos de ciclo de vida. La visión del modelo cascada del desarrollo de software es muy simple; dice que el desarrollo de software puede ser a través de una secuencia simple de fases. Cada fase tiene un conjunto de metas bien definidas, y las actividades dentro de una fase contribuyen a la satisfacción de metas de esa fase o quizás a una subsecuencia de metas de la fase. Las flechas muestran el flujo de información entre las fases. La flecha de avance muestra el flujo normal. Las flechas hacia atrás representan la retroalimentación.

El modelo de ciclo de vida cascada, captura algunos principios básicos:

- Planear un proyecto antes de embarcarse en él.
- Definir el comportamiento externo deseado del sistema antes de diseñar su arquitectura interna.
- Documentar los resultados de cada actividad.
- Diseñar un sistema antes de codificarlo.
- Testear un sistema después de construirlo.



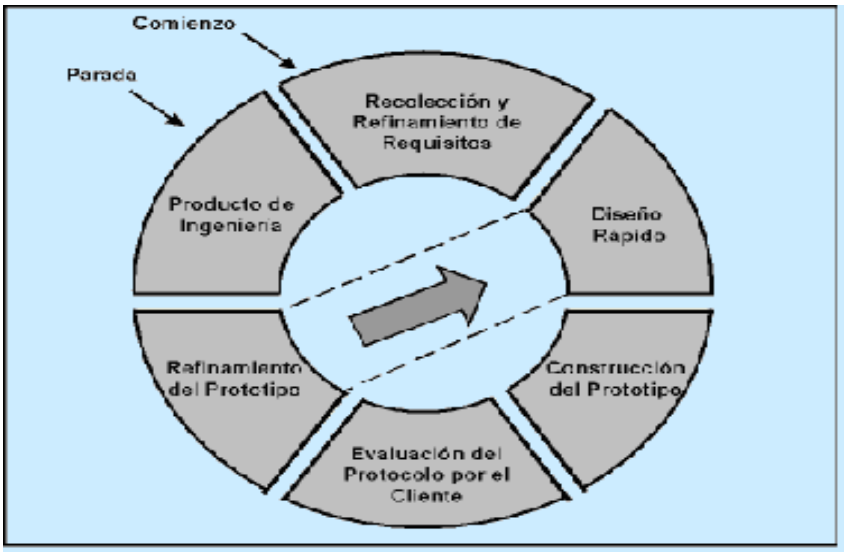
Modelo de Ciclo de Vida Cascada.

2.1.2 Construcción de prototipos

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:

- 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;
- un prototipo que implemente algunos subconjuntos de la función requerida del programa deseado,
- 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.

Aunque pueden aparecer problemas, la construcción de prototipos es un paradigma efectivo para la ingeniería del software. La clave está en definir al comienzo las reglas del juego; esto es, el cliente y el técnico deben estar de acuerdo en que el prototipo se construya para servir sólo como un mecanismo de definición de los requisitos.

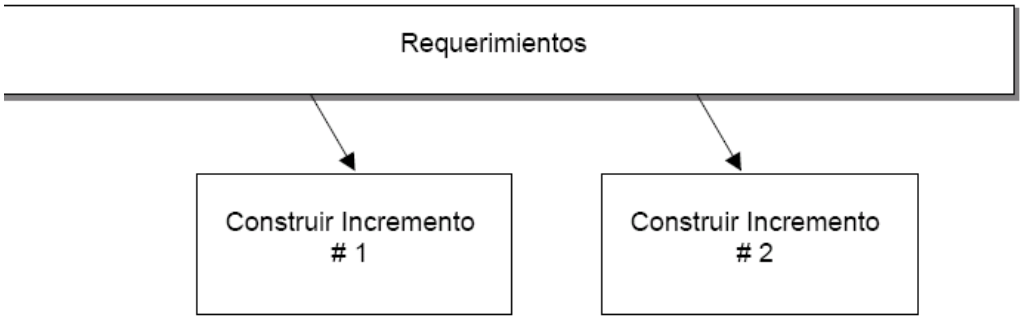


Creación de Prototipos

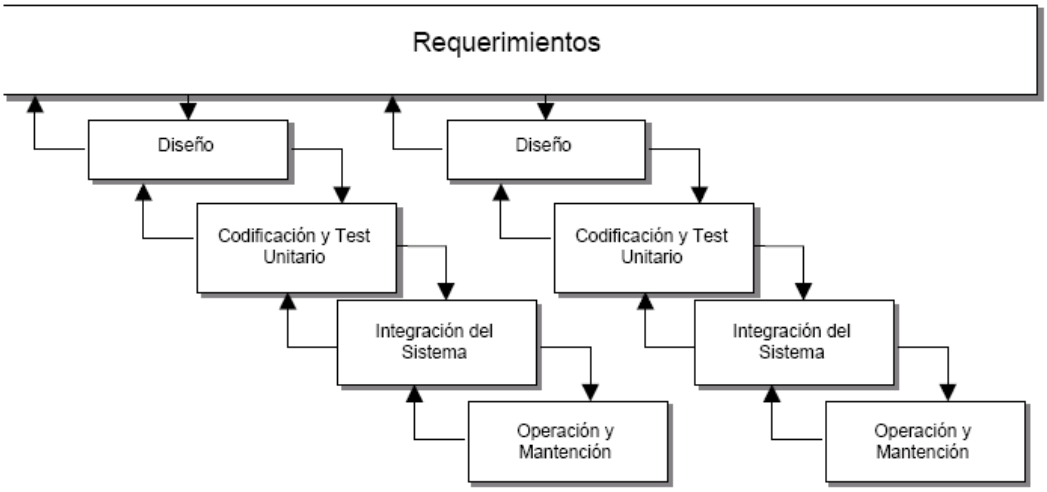
2.1.3 Modelo de Desarrollo Incremental.

Los riesgos asociados con el desarrollo de sistemas largos y complejos son enormes. Una forma de reducir los riesgos es construir sólo una parte del sistema, reservando otros aspectos para niveles posteriores. El desarrollo incremental es el proceso de construcción siempre incrementando subconjuntos de requerimientos del sistema. Típicamente, un documento de requerimientos es escrito al capturar todos los requerimientos para el sistema completo. Note que el desarrollo incremental es 100% compatible con el modelo cascada. El desarrollo incremental no demanda una forma específica de observar el desarrollo de algún otro incremento. Así, el modelo cascada puede ser usado para administrar cada esfuerzo de desarrollo, como se muestra en la figura. El modelo de desarrollo incremental provee algunos beneficios significativos para los proyectos:

- Construir un sistema pequeño es siempre menos riesgoso que construir un sistema grande.
- Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requerimientos planeados para los niveles subsiguientes son correctos.
- Si un error importante es realizado, sólo la última iteración necesita ser descartada.
- Reduciendo el tiempo de desarrollo de un sistema (en este caso en incremento del sistema) decrecen las probabilidades que esos requerimientos de usuarios puedan cambiar durante el desarrollo.
- Si un error importante es realizado, el incremento previo puede ser usado.
- Los errores de desarrollo realizados en un incremento, pueden ser arreglados antes del comienzo del próximo incremento.



Modelo de Desarrollo Incremental.



Modelo de Desarrollo Incremental con desarrollo en cascada de los incrementos.

2.1.4 Modelo Espiral.

El modelo espiral de los procesos software es un modelo del ciclo de *meta-vida*. En este modelo, el esfuerzo de desarrollo es iterativo. Tan pronto como uno completa un esfuerzo de desarrollo, otro comienza. Además, en cada desarrollo ejecutado, puedes seguir estos cuatros pasos:

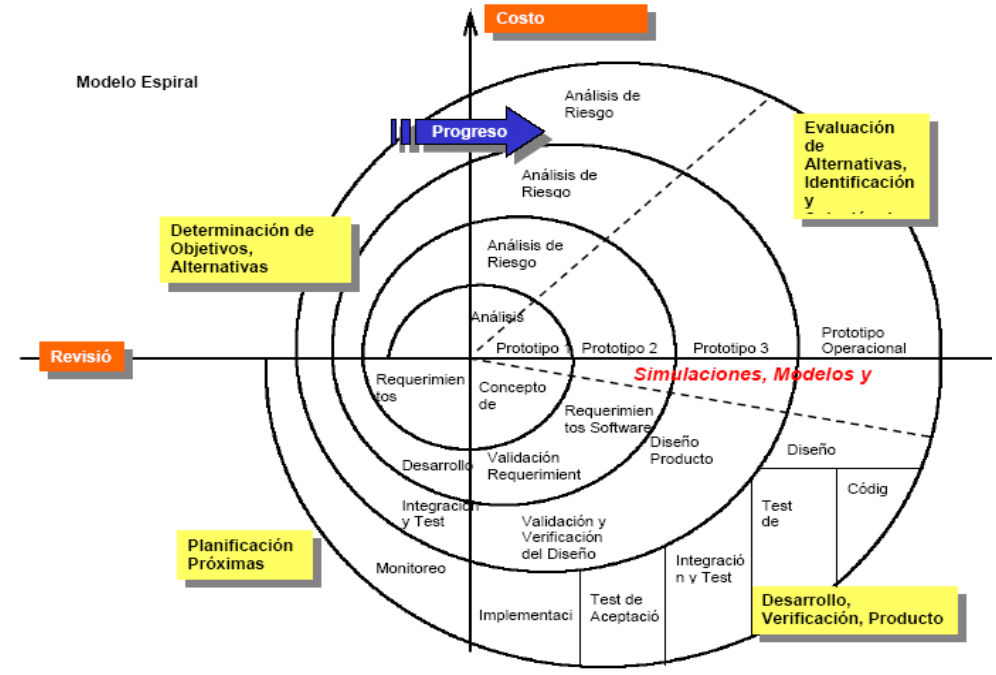
1. Determinar qué se quiere lograr.
2. Determinar las rutas alternativas que se pueden tomar para lograr estas metas. Por cada una, analizar los riesgos y resultados finales, y seleccionar la mejor.
3. Seguir la alternativa seleccionada en el paso 2.
4. Establecer qué se tiene terminado.

La dimensión radial en la figura refleja costos acumulativos incurridos en el proyecto. Observemos un escenario particular. Digamos que en este proyecto, nosotros

viajaremos a resolver un conjunto particular de problemas del cliente. Durante el primer viaje alrededor de la espiral, analizamos la situación y determinamos que los mayores riesgos son la interfaz del usuario. Después de un cuidadoso análisis de las formas alternativas de direccionar esto (por ejemplo, construir un sistema y esperar lo mejor, escribir una especificación de requerimientos y esperar que el cliente lo entienda, y construir un prototipo), determinamos que el mejor curso de acción es construir un prototipo. Lo realizamos, luego proveemos el prototipo al cliente quien nos provee con retroalimentación útil. Ahora, comenzamos el segundo viaje alrededor de la espiral. Este tiempo decidimos que el mayor riesgo es ese miedo a que muchos nuevos requerimientos comiencen a aparecer sólo después de que el sistema sea desplegado. Analicemos las rutas alternativas, y decidimos que la mejor aproximación es construir un incremento del sistema que satisfaga sólo los requerimientos mejor entendidos. Hagámoslo ya. Después del despliegue, el cliente nos provee de retroalimentación que dirá si estamos correctos con esos requerimientos, pero 50 nuevos requerimientos ahora se originarán en las cabezas de los clientes. Y el tercer viaje alrededor de la espiral comienza. El modelo espiral captura algunos principios básicos:

- Decidir qué problema se quiere resolver antes de viajar a resolverlo.
- Examinar tus múltiples alternativas de acción y elegir una de las más convenientes.
- Evaluar qué tienes hecho y qué tienes que haber aprendido después de hacer algo.
- No ser tan ingenuo para pensar que el sistema que estás construyendo será “EL” sistema que el cliente necesita, y
- Conocer (comprender) los niveles de riesgo, que tendrás que tolerar.

El modelo espiral no es una alternativa del modelo cascada, ellos son completamente compatibles.



2.1.5 Modelo Concurrente.

Como el modelo espiral, el modelo concurrente provee una meta-descripción del proceso software. Mientras que la contribución primaria del modelo espiral es en realidad que esas actividades del software ocurran repetidamente, la contribución del modelo concurrente es su capacidad de describir las múltiples actividades del software ocurriendo simultáneamente. Esto no sorprende a nadie que ha estado involucrado con las diversas actividades que ocurren en algún tiempo del proceso de desarrollo de software. Discutamos un poco tales casos: Los requerimientos son usualmente "líneas de base", cuando una mayoría de los requerimientos comienzan a ser bien entendidos, en este tiempo se dedica un esfuerzo considerable al diseño. Sin embargo, una vez que comienza el diseño, cambios a los requerimientos son comunes y frecuentes (después de todo, los problemas reales cambian, y nuestro entendimiento de los problemas desarrollados también). Es desaconsejado detener el diseño en este camino cuando los requerimientos cambian; en su lugar, existe una necesidad de modificar y rehacer líneas de base de los requerimientos mientras progresa el diseño. Por supuesto, dependiendo del impacto de los cambios de los requerimientos el diseño puede no ser afectado, medianamente afectado o se requerirá comenzar todo de nuevo. Durante el diseño de arquitectura, es posible que algunos componentes comiencen a ser bien definidos antes que la arquitectura completa sea estabilizada. En tales casos, puede ser posible comenzar el diseño detallado en esos componentes estables. Similarmente, durante el diseño detallado, puede ser posible proceder con la codificación y quizás regular testeando en forma unitaria o realizando testeo de integración previo a llevar a cabo el diseño detallado de todos los componentes.
En algunos proyectos, múltiples etapas de un producto se han desarrollado concurrentemente. Por ejemplo, no es inusual estar haciendo mantención de la etapa 1 de un producto, y al mismo tiempo estar haciendo mantención sobre un componente 2, mientras que se está haciendo codificación sobre un componente 3, mientras se realiza diseño sobre una etapa 4, y especificación de requisitos sobre un componente 5. En todos estos casos, diversas actividades están ocurriendo simultáneamente. Eligiendo seguir un proyecto usando técnicas de modelación concurrente, se posibilita el conocimiento del estado verdadero en el que se encuentra el proyecto.

2.1.6 Proceso Unificado Rational (RUP)

Es un proceso de desarrollo de software de forma disciplinada de asignar tareas y responsabilidades en una empresa de desarrollo (quién hace qué, cuándo y cómo).

Objetivos:
Asegurar la producción de software de calidad dentro de plazos y presupuestos predecibles. Dirigido por casos de uso, centrado en la arquitectura, iterativo (mini-proyectos) e incremental (versiones).

Es también un producto:

- Desarrollado y mantenido por Rational.
- Actualizado constantemente para tener en cuenta las mejores prácticas de acuerdo con la experiencia.

Aumenta la productividad de los desarrolladores mediante acceso a: Base de conocimiento, plantillas y herramientas.

Se centra en la producción y mantenimiento de modelos del sistema más que en producir documentos.

RUP es una guía de cómo usar UML de la forma más efectiva.
Existen herramientas de apoyo a todo el proceso:

- Modelamiento visual, programación, pruebas, etc.

Ciclos y fases

RUP divide el proceso de desarrollo en ciclos, teniendo un producto al final de cada ciclo.

Cada ciclo se divide en cuatro Fases:

Inicio
Elaboración
Construcción
Transición

Cada fase concluye con un hito bien definido donde deben tomarse ciertas decisiones.

Fase: Inicio

- Se establece la oportunidad y alcance el proyecto.
- Se identifican todas las entidades externas con las que se trata (actores) y se define la interacción a un alto nivel de abstracción:
  - Identificar todos los casos de uso
  - Describir algunos en detalle
- La oportunidad del negocio incluye:
  - Criterios de éxito
  - Identificación de riesgos
  - Estimación de recursos necesarios
  - Plan de las fases incluyendo hitos

Productos:

- Un documento de visión general:
  - Requisitos generales del proyecto
  - Características principales
  - Restricciones
- Modelo inicial de casos de uso (10% a 20 % listos).
- Caso de negocio:
  - Contexto
  - Criterios de éxito
  - Pronóstico financiero
- Identificación inicial de riesgos.

- Plan de proyecto.
- Uno o más prototipos
- Glosario.

Fase: Elaboración

- Objetivos:
  - Analizar el dominio del problema
  - Establecer una arquitectura base sólida
  - Desarrollar un plan de proyecto
  - Eliminar los elementos de mayor riesgo para el desarrollo exitoso del proyecto

- Visión de “un kilómetro de amplitud y un centímetro de profundidad” porque las decisiones de arquitectura requieren una visión global del sistema.

Productos:

- Es la parte más crítica del proceso:
  - Al final toda la ingeniería “dura” está hecha
  - Se puede decidir si vale la pena seguir adelante
- A partir de aquí la arquitectura, los requisitos y los planes de desarrollo son estables.
- Ya hay menos riesgos y se puede planificar el resto del proyecto con menor incertidumbre.
- Se construye una arquitectura ejecutable que contemple:
  - Los casos de uso críticos
  - Los riesgos identificados
- Modelo de casos de uso (80% completo) con descripciones detalladas.
- Otros requisitos no funcionales o no asociados a casos de uso.
- Descripción de la Arquitectura del Software.
- Un prototipo ejecutable de la arquitectura.
- Lista revisada de riesgos y del caso de negocio.
- Plan de desarrollo para el resto del proyecto.
- Un manual de usuario preliminar.

Fase: Construcción

- En esta fase todas las componentes restantes se desarrollan e incorporan al producto.
- Todo es probado en profundidad.
- El énfasis está en la producción eficiente y no ya en la creación intelectual.
- Puede hacerse construcción en paralelo, pero esto exige una planificación detallada y una arquitectura muy estable.

Productos:

- El producto de software integrado y corriendo en la plataforma adecuada.
- Manuales de usuario.
- Una descripción del “release” actual.

Fase: Transición

- El objetivo es traspasar el software desarrollado a la comunidad de usuarios.
- Una vez instalado surgirán nuevos elementos que implicarán nuevos desarrollos (ciclos).
- Incluye:
  - Pruebas Beta para validar el producto con las expectativas del cliente
  - Ejecución paralela con sistemas antiguos
  - Conversión de datos
  - Entrenamiento de usuarios
  - Distribuir el producto

Objetivos:

- Obtener autosuficiencia de parte de los usuarios.
- Concordancia en los logros del producto de parte de las personas involucradas.
- Lograr el consenso cuanto antes para liberar el producto al mercado.

2.1.7 Seleccionando Modelos de Ciclo de Vida.

Los modelos presentados, suministran una guía con el fin de ordenar las diversas actividades técnicas en el proyecto de desarrollo de software e intentan suministrar un marco para la administración en el desarrollo y el mantenimiento. Aunque todos ellos son compatibles unos con otros, un proyecto puede decidir cuáles enfoques son más útiles en situaciones especiales. Criterios a considerar:

- Madurez de la aplicación (relacionado a la probabilidad que muchos requerimientos comenzarán a conocerse sólo después del uso del sistema).
- Complejidad del problema y de la solución.
- Frecuencias y magnitudes esperadas de los cambios de requerimientos.
- Financiamiento disponible, y su perfil como una función del tiempo.
- Acceso de los desarrolladores a los usuarios.
- Certeza de requerimientos conocidos.

Otros que pueden incluirse:

- Tolerancia al riesgo.
- Planes y presupuestos críticos
- Grado de lentitud de construcción dentro de los planes y presupuestos.

Considerando la importancia de la planificación se recomienda realizar el desarrollo de un proyecto de software bajo el modelo espiral insertando en él, cualquier otro modelo que se requiera dependiendo de las necesidades que se presenten. Este modelo permite realizar una planificación del proceso de desarrollo del software considerando los riesgos asociados en cada etapa identificada. El identificar los riesgos en proyectos, evaluar su impacto, monitorear y controlar el avance del desarrollo del proyecto, permite al administrador aumentar las posibilidades de éxito de un proyecto o, minimizar las posibilidades de fracaso de éste. Uno de los factores que más influyen en el proceso de desarrollo de software y que prácticamente acompaña a toda aplicación es el hecho de que en su mayoría, no hay forma de tener todos los requerimientos corregidos antes del desarrollo del software. Muchas veces los requerimientos emergen

a medida que la aplicación o partes de ella están disponible para experimentación práctica. En todos los casos, el trabajo comienza con la determinación de objetivos, alternativas y restricciones, paso que a veces se llama recolección preliminar de requisitos.

El prototipado es ampliamente recomendado para realizar la especificación de requerimientos, se debe notar que la idea del prototipado es capturar por retroalimentación los objetivos, necesidades y expectativas del cliente por lo cual no se debe caer en una utilización de estos prototipos como partes finales del sistema, ya que en su desarrollo generalmente no se consideran aspectos de calidad, ni otros asociados con facilitar la etapa de mantención del sistema. El prototipo trata de minimizar los cambios en los requerimientos, mientras que el diseño modular (incremental, en funcionalidades) trata de minimizar el impacto de los cambios en los requerimientos.

El cambio es una propiedad intrínseca del software. Hoy en día el software debe poseer un enfoque evolutivo, un sistema debe evolucionar para acomodar la naturaleza evolutiva de los grandes sistemas. El software cambia constantemente, debido a la necesidad de reparar el software (eliminando errores no detectados anteriormente) como a la necesidad de apoyar la evolución de los sistemas a medida que aparecen nuevos requerimientos o cambian los antiguos. Por lo cual es importante enfatizar que no tiene sentido entonces que un proyecto tome estrictamente una decisión concerniente con cual modelo se adherirá. Los modelos de ciclo de vida presentados, son complementarios en vez de excluyentes.

En muchos casos, los paradigmas pueden y deben combinarse de forma que puedan utilizarse las ventajas de cada uno en un único proyecto. El paradigma del modelo en espiral lo hace directamente, combinando la creación de prototipos y algunos elementos del ciclo de vida clásico, en un enfoque evolutivo para la ingeniería de software. No hay necesidad por tanto de ser dogmático en la elección de los paradigmas para la ingeniería de software: la naturaleza de la aplicación debe dictar el método a elegir.

2.2 MÉTRICAS

La única forma de probar la calidad es establecer cuál es el nivel actual y disponer un plan de mejoramiento. Hay ciertos aspectos importantes del software que pueden ser medidos. En términos del software mismo (medidas de producto), existen métricas de complejidad, tamaño, de flujo de información, y muchas más

El software al ser intangible, no tener peso, ni volumen, ni superficie, etc. se mide a través de diversos aspectos clave en el desarrollo. La medición y estimación atacan los tres problemas claves de la ingeniería del software:

1- Estimar costos y recursos en un proyecto software
2- Garantizar la calidad del producto final
3- Mejorar la productividad del ingeniero de software durante el desarrollo.

Para estimar los recursos es necesario tener en cuenta una serie de factores de riesgo que influyen sustancialmente en la precisión de las estimaciones de los recursos humanos necesarios para la realización del proyecto. Los más importantes son:

*Complejidad de la tarea.
*Modificaciones permitidas a lo largo del desarrollo
*Experiencia previa de los desarrolladores
*Duración fijada del proyecto.
*Estructuración del problema y de las tareas.
*Disponibilidad de datos e información suministrada por el usuario.
*Disponibilidad y facilidad de comunicación con el usuario.

Además de las fases estándar del desarrollo, hay que tener en cuenta la coordinación y seguimiento del proyecto que suponen una importante carga de trabajo y que son olvidadas durante la planificación o no se le dedica mucho.

El costo global se compone de las partidas de viajes, hardware (nuevo o actualización), software (en caso de comprar algún paquete para el desarrollo), gastos comunes, y personal que es el mas influyente, ya que el costo de un proyecto es directamente proporcional a los recursos humanos.

El proceso engloba todas las actividades y fases que se llevan a cabo durante la realización del proyecto. Se persigue determinar si en cada fase los resultados producidos se corresponden con los esperados y en establecer un control sobre los recursos estimados para cada una de las fases.

El producto incluye cualquier documento o software desarrollado que se genere durante el proceso completo. En las medidas de productos software existen medidas directas (costo del proyecto, esfuerzo empleado, líneas de código implementadas, etc.) y medidas indirectas (funcionalidad, fiabilidad, eficiencia, facilidad de mantenimiento, etc.).

2.3 Proyecto de Desarrollo de Software

Los sistemas de información basados en computadoras sirven para diversas finalidades que van desde el procesamiento de las transacciones de una empresa hasta proveer de la información necesaria para decidir sobre asuntos que se presentan con frecuencia.

En algunos casos los factores que deben considerarse en un proyecto de sistema de información, como el aspecto más apropiado de la computadora o la tecnología de comunicaciones que se va a utilizar, el impacto del nuevo sistema sobre los empleados de la empresa y las características específicas que el sistema debe tener se pueden determinar de manera secuencial

Antes de considerar cualquier investigación de sistemas, la solicitud de proyecto debe examinarse para determinar con precisión lo que el solicitante desea; ya que muchas solicitudes que provienen de empleados y usuarios no están formuladas de manera clara.

Estudio de factibilidad

En la investigación preliminar un punto importante es determinar que el sistema solicitado sea factible. Existen tres aspectos relacionados con el estudio de factibilidad, que son realizados por lo general por analistas capacitados o directivos.

Factibilidad técnica.
Estudia si el trabajo para el proyecto, puede desarrollarse con el software y el personal existente, y si en caso de necesitar nueva tecnología, cuales son las posibilidades de desarrollarla (no solo el hardware).

Factibilidad económica.
Investiga si los costos se justifican con los beneficios que se obtienen, y si se ha invertido demasiado, como para no crear el sistema si se cree necesario.

Factibilidad operacional.
Investiga si será utilizado el sistema, si los usuarios usaran el sistema, como para obtener beneficios.

Algunas organizaciones reciben tantas solicitudes de sus empleados que sólo es posible atender unas cuantas. Sin embargo, aquellos proyectos que son deseables y factibles deben incorporarse en los planes. En algunos casos el desarrollo puede comenzar inmediatamente, aunque lo común es que los miembros del equipo de sistemas estén ocupados en otros proyectos. Cuando esto ocurre, la administración decide que proyectos son los más importantes y el orden en que se llevarán acabo.

Después de aprobar la solicitud de un proyecto se estima su costo, el tiempo necesario para terminarlo y las necesidades de personal

CAPÍTULO III

3.1 REVISIÓN DE LOS PRINCIPALES PROCESOS DE LA I.S.

3.1.1 Análisis de Requisitos

El Análisis de Requisitos es un proceso de descubrimiento, refinamiento, modelización y especificación. Tanto el desarrollador como el cliente juegan un papel activo en la especificación y el Análisis de Requisitos. El cliente intenta volver a plantear detalladamente el concepto de la función y del comportamiento del Software. El desarrollador actúa como interrogador, como consultor y como persona que resuelve problemas. El análisis y la Especificación de los Requisitos puede parecer una tarea relativamente sencilla, pero las apariencias engañan. El contenido de la comunicación es muy denso. Abundan los casos en que se pueden llegar a malas interpretaciones o falta de información

3.1.2 Análisis y Diseño del Software

El análisis del software es un conjunto o disposición de procedimientos o programas relacionados de manera que juntos forman una sola unidad. Un conjunto de hechos, principios y reglas clasificadas y dispuestas de manera ordenada mostrando un plan lógico en la unión de las partes. Un método, plan o procedimiento de clasificación para hacer algo. También es un conjunto o arreglo de elementos para realizar un objetivo predefinido en el procesamiento de la Información. Esto se lleva a cabo teniendo en cuenta ciertos principios:

- Debe presentarse y entenderse el dominio de la información de un problema.
- Defina las funciones que debe realizar el Software.
- Represente el comportamiento del software a consecuencias de acontecimientos externos.
- Divida en forma jerárquica los modelos que representan la información, funciones y comportamiento.

El proceso debe partir desde la información esencial hasta el detalle de la Implementación.

La función del Análisis puede ser dar soporte a las actividades de un negocio, o desarrollar un producto que pueda venderse para generar beneficios.

El diseño del software requiere de precisión y de creatividad, por parte del diseñador; su propósito es: Especificar la estructura interna y los detalles de procesamiento de un sistema y proporcionar un ensayo de revisión del por que fueron timadas las decisiones de diseño.

El objetivo del diseñador es producir un modelo o representación de una entidad que se construirá más adelante.

3.1.3 Pruebas del Software

En término generales, las pruebas se definen como el proceso de localizar y reparar los errores. La fase de prueba en el desarrollo del software es vista como la actividad anterior a la presentación del producto final.

Dado que una prueba completa no es viable, la pregunta de que tantas pruebas y en donde deben hacerse depende en un alto grado de la aplicación en particular, por ejemplo en un simple paquete de procesador de palabras no requiere de muchas pruebas rigurosas, caso contrario sería en un sistema complejo como los utilizados en las áreas militares, en investigaciones científicas, en los hospitales, etc.

3.1.4 Calidad del Software

Se puede decir que es la concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente.

Esta definición de calidad sirve para hacer hincapié en los siguientes puntos:
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.

Los estándares específicos 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.

Existe un conjunto de “requisitos implícitos” que a menudo no se mencionan. Por ejemplo: El deseo de un buen mantenimiento. Si el Software se ajusta a sus requisitos explícitos pero falla en alcanzar los requisitos implícitos, la Calidad del Software 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.

3.1.5 Configuración

Esta es la actividad esencial de calidad para mantener el control de la actividad y el estado de todo el hardware, documentos y software.

El control de las modificaciones es de importancia capital para el desarrollo de software puesto que no existe visibilidad para el software en disco mas que por sus etiquetas y documentación.

3.1.6 Documentación del Software

Dado que la única forma visible del software es la documentación, una estructura formal que mantenga ligados los documentos de especificación y diseño y listados de código es esencial. Por supuesto, estas ligas deben ser apropiadas para el tamaño del paquete de software.

3.1.7 Mantenimiento del Software

El software sufrirá cambios después de que se entregue al cliente. Los cambios ocurrirán debido a posibles errores que se hayan encontrado, a que el software debe adaptarse a cambios en el entorno externo, o debido a que el cliente requiera ampliaciones funcionales o de rendimiento. Posteriormente, ha de ser descartado y debe construirse el software real, con los ojos puesto en la calidad y en el mantenimiento.

3.2 LA GESTIÓN DEL PROYECTO DE SOFTWARE

La gestión del proyecto de software es el primer nivel del proceso de ingeniería de software, porque cubre todo el proceso de desarrollo. Para conseguir un proyecto de software fructífero se debe comprender el ámbito del trabajo a realizar, los riesgos en los que se puede incurrir, los recursos requeridos, las tareas a llevar a cabo, el esfuerzo (costo) a consumir y el plan a seguir.

Gestión son todas las actividades y tareas ejecutadas por una o más personas con el propósito de planificar y controlar las actividades de otros para alcanzar un objetivo o completar una actividad que no puede ser realizada por otros actuando independientemente.

Planificación de un Proyecto de Ingeniería de Software.

La planificación involucra la especificación de objetivos y metas para un proyecto y las estrategias, políticas, planes y procedimientos para alcanzarlos. Todo proyecto de ingeniería de software debe partir con un buen plan. La planificación es necesaria por la existencia de incertezas sobre el ambiente del proyecto software y sobre fuentes externas. La planificación enfoca su atención en las metas del proyecto, riesgos potenciales y problemas que puedan interferir con el cumplimiento de esas metas.

Los principales problemas en la planificación de un proyecto de ingeniería de software incluyen los siguientes:

- Requerimientos incorrectos e incompletos.
- Muchas especificaciones de requerimientos son inestables y sujetas a cambios mayores.
- La planificación no se lleva a cabo por la creencia errónea de que es una pérdida de tiempo y los planes cambiarán de todos modos.
- La planificación de costos y plazos no es actualizada y se basa en necesidades de mercadeo y no de los requerimientos del sistema.
- Es difícil estimar el tamaño y complejidad del proyecto de software de modo de realizar una estimación de costos y plazos realista.
- Los costos y plazos no son re estimados cuando los requerimientos del sistema o el ambiente de desarrollo cambia.
- No se manejan factores de riesgo.
- La mayoría de las organizaciones de desarrollo de software no recolectan datos de proyectos pasados.
- Las compañías no establecen políticas o procesos de desarrollo de software.

Actividades que se derivan de la planificación.

- Fijar los objetivos y metas
- Desarrollar estrategias
- Desarrollar políticas
- Anticipar futuras situaciones
- Conducir un establecimiento de riesgos
- Determinar posibles cursos de acción
- Tomar decisiones de planificación
- Fijar procedimientos y reglas
- Desarrollar los planes del proyecto
- Preparar presupuestos
- Documentar los planes del proyecto.

3.3 ESTIMACIÓN DE TIEMPOS Y RECURSOS

Una de las actividades cruciales del proceso de gestión es la planificación, la cual se basa en una buena estimación del esfuerzo requerido para realizar el proyecto, duración cronológica del proyecto y el costo (en miles de soles o dólares).

La técnica más utilizada para realizar estimaciones de costos y plazos es la que denomino “juicio experto”, donde el administrador del proyecto recurre a alguien que haya desarrollado aplicaciones similares para que realice una estimación de los recursos y tiempo a necesitar para el desarrollo. Otra técnica muy natural es utilizar descomposición. Esto es, dividir el problema en partes más pequeñas y estimar cada una por separado, utilizando un juicio experto o algún método más formal (como estimar sobre la base del tamaño en una métrica formal).

Además, se suele realizar una estimación optimista (EO), otra más probable (EMP) y una pesimista (EP), y asignarle una probabilidad a cada una, obteniendo así nuestra estimación mediante:

$$E=EO *Po + EMP*Pmp + EP*Pp$$

Donde Po es la probabilidad asignada a la estimación optimista, Pmp la asignada a la más probable y Pp la asignada a la pesimista.

Así, por ejemplo, dado un proyecto software X, estimamos que lo mínimo que nos podríamos demorar son 5 meses con probabilidad de un 10%, y lo máximo de 12 meses con probabilidad de un 30%, y seguramente nos tardaremos 10 meses. Por otro lado, lo mínimo que nos costará es US$2000 con probabilidad 15%, lo máximo US$6200 con probabilidad 20%, y lo más probable es que el costo sea de US$4000. Con estos datos podemos obtener dos estimaciones:

Tiempo = 5 *0.1 + 12 * 0.3 + 10 * 0.6 =0.5 + 3.6 +6 = 10.1 meses

Costo = 2000 * 0.15 + 6200*0.2 + 4000 * 0.65 = 300 + 1240 + 2600 = US$4140

Recursos Humanos

El planificador comienza evaluando el ámbito y seleccionando las habilidades técnicas que se requieren para llevar a cabo el desarrollo.

Recursos del Hardware

Durante la planificación del proyecto de Software se deben considerar tres categorías de Hardware:

- Sistema de desarrollo. Denominado sistema anfitrión; compuesto por la computadora a utilizarse en la fase de desarrollo del Software y sus periféricos asociados
- Máquina objetivo. Puede que se utilice como sistema de desarrollo porque soporta múltiples usuarios.
- Los demás elementos de Hardware del nuevo sistema

Recursos de Software

La primera aplicación que se le dio al software en el desarrollo del software, fue lo que se denominó RECONSTRUCCIÓN, se comenzó escribiendo un primitivo traductor de lenguaje ensamblador a lenguaje de máquina y se usó para desarrollar un ensamblador mas sofisticado, los equipos de desarrollo fueron reconstruyendo eventualmente el software hasta llegar a construir compiladores de lenguaje de alto nivel y otras herramientas.

3.4 MÉTRICAS

La medición es muy común en el mundo de la ingeniería. Se miden potencias, consumos, pesos, fuerzas, voltajes, niveles de ruido, etc. Desgraciadamente, la medición no es una práctica común en el mundo de la ingeniería de software.

Razones que justifican la medición del software son

(a) para indicar la calidad del producto,
(b) para evaluar la productividad de la gente que desarrolla el producto,
(c) para evaluar los beneficios (en términos de productividad y calidad) derivados del uso de nuevos métodos y herramientas de ingeniería de software,
(d) para establecer una línea base para la estimación y
(e) para ayudar a justificar el uso de nuevas herramientas o formación adicional.

Las mediciones del mundo físico pueden englobarse en dos categorías: medidas directas (el largo de un tornillo) y medidas indirectas (la calidad de los tornillos producidos). Las métricas del software pueden ser catalogadas en forma análoga. Entre las medidas directas del proceso de ingeniería de software se encuentra el costo y el esfuerzo aplicado. Entre las medidas directas del producto se encuentran las líneas de código producidas, velocidad de ejecución y los defectos observados en un período de tiempo. Entre las medidas indirectas se encuentran la calidad, funcionalidad, eficiencia, facilidad de mantenimiento, etc.

Las métricas de software se pueden clasificar como métricas orientadas a la función o métricas orientadas al tamaño. También se pueden clasificar según la información que entregan: métricas de productividad, las que se centran en el rendimiento del proceso de ingeniería de software, métricas de calidad, proporcionan una indicación de cómo se ajusta el software a los requisitos explícitos e implícitos del cliente y las métricas técnicas, que se centran más en el software que en el proceso a través del cuál se ha desarrollado (por ejemplo grado de modularidad o grado de complejidad lógica).

Para resumir, podemos decir que una métrica de software es una función cuyas entradas son datos del software (o el proceso del software) y cuya salida es un valor numérico único, que puede ser interpretado como el grado en que el software (o el proceso del software) posee un atributo dado.

3.5 PROGRAMACIÓN DE PROYECTOS CON PERT - CPM

Un proyecto define una combinación de actividades interrelacionadas que deben ejecutarse en un cierto orden antes que el trabajo completo pueda terminarse. Las actividades están interrelacionadas en una secuencia lógica en el sentido que algunas de ellas no pueden comenzar hasta que otras se hayan terminado. Una actividad en un proyecto, usualmente se ve como un trabajo que requiere tiempo y recursos para su terminación.

Para la administración de proyectos (planeación, programación y control de proyectos) existen dos técnicas analíticas: PERT (program evaluation and review technique - técnica de evaluación y revisión de proyectos) y CPM (critical path method - método de ruta critica).

La programación de proyectos por PERT - CPM consiste en:

Planeación
Se descompone el proyecto en actividades distintas, luego se determina la estimación de tiempo para las actividades, construyendo un diagrama de red.

Programación
Permite construir un diagrama de tiempo que muestre los tiempos de iniciación y terminación para cada actividad. Así como su relación con otras actividades del proyecto. Además señalar las actividades criticas.

Control
Usando diagramas de flechas y gráficas de tiempo se hacen reportes periódicos del progreso, que permiten actualizar la red.

Reglas para construir el diagrama de flechas

Regla1.- Cada actividad esta representada por una y solamente una fecha en la red.

Regla2.- Dos actividades diferentes no pueden identificarse por los mismos eventos terminales y de comienzo.

Regla3.- A fin de asegurar la relación de precedencia correcta en el diagrama de flechas, las siguientes preguntas deben responderse cuando se agrega cada actividad a la red:

1. ¿Qué actividades deben terminarse inmediatamente antes de que esta actividad pueda comenzar?
2. ¿Qué actividades deben seguir a esta actividad?
3. ¿Qué actividades deben efectuarse concurrentemente con esta actividad?

Ejercicio
Construya el diagrama de flechas que comprende las actividades: A, B, C, D, ….., L

- A, B y C las primeras actividades del proyecto pueden comenzar simultáneamente
- A y B preceden a D
- B precede a E, F y H
- F y C preceden a G
- E y H preceden a I y J
- C, D, F y J preceden a K
- K precede a L
- I, G y L son las actividades terminales del proyecto.

Cálculos de ruta critica

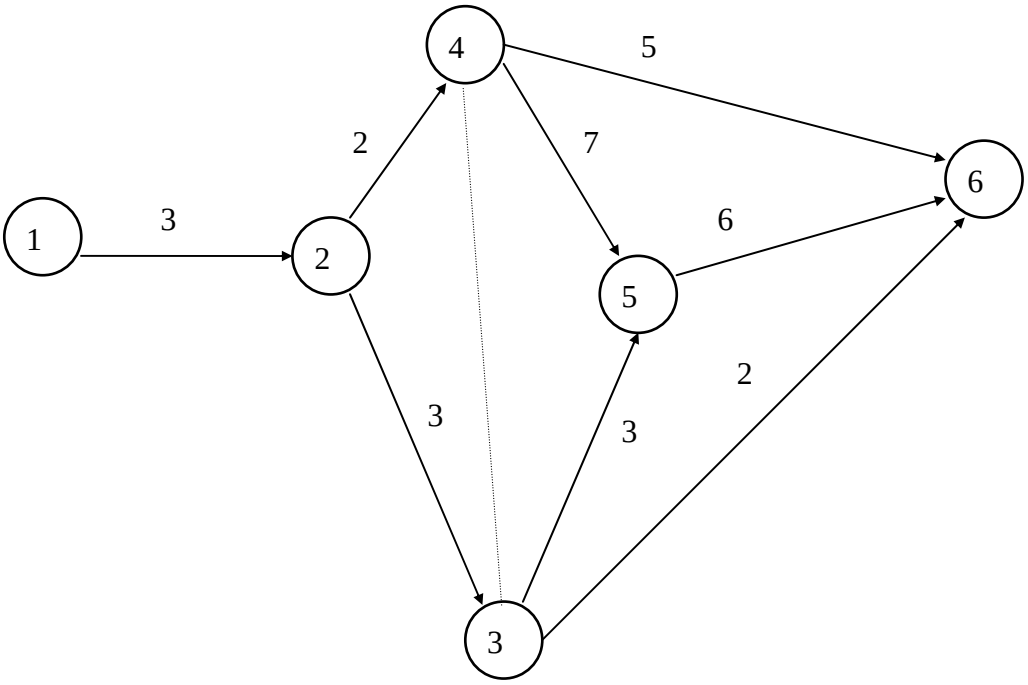
La aplicación de PERT - CPM deberá proporcionar un programa, especificando las flechas de inicio y terminación de cada actividad. El resultado final es clasificar las actividades de los proyectos como:

Criticas.- Una demora en su comienzo causará una demora en la fecha de terminación del proyecto.

No crítica.- Es tal que el tiempo entre su comienzo más temprano y su terminación más tarde es más grande que su duración actual. En este caso, se dice que la actividad no crítica tiene un **tiempo de holgura.**

Ejercicio

Considere la siguiente red, donde el tiempo requerido para cada actividad se indica sobre las flechas y determine la ruta crítica



CAPÍTULO IV

4.1 ANÁLISIS DE REQUISITOS

Para que un esfuerzo de desarrollo de software tenga éxito, es esencial comprender perfectamente los requisitos del Software. Independientemente de que bien diseñado o codificado que esté un programa, si se ha analizado y especificado pobremente, decepcionará al usuario y desprestigiará al que lo ha desarrollado.

El Análisis de Requisitos es un proceso de descubrimiento, refinamiento, modelización y especificación.

Tanto el desarrollador como el cliente juegan un papel activo en la especificación y el Análisis de Requisitos. El cliente intenta volver a plantear detalladamente el concepto de la función y del comportamiento del Software. El desarrollador actúa como interrogador, como consultor y como persona que resuelve problemas.

El análisis y la Especificación de los Requisitos puede parecer una tarea relativamente sencilla, pero las apariencias engañan. El contenido de la comunicación es muy denso. Abundan los casos en que se pueden llegar a malas interpretaciones o falta de información. Es probable que se llegue a ambigüedades.

Requerimientos del software

La recopilación y el análisis de los requerimientos del sistema es una de las fases más importantes en un proyecto para el alcance del éxito.

Alcance de los requerimientos

Su objetivo principal es establecer un acuerdo entre el usuario y el analista sobre qué debe hacer el software, pero el rol exacto de la Especificación de Requerimientos en el proceso de desarrollo depende del ciclo de vida que se emplee cumple con los siguientes objetivos:

- Proporcionar la primera entrada para la fase de diseño.
- Trazar las líneas en las que las pruebas de aceptación se lleven a cabo.

La preparación de una Especificación adecuada de Requerimientos reduce los costos y el riesgo general asociado con el desarrollo; esto es que el costo de corregir o modificar se reduce una vez instalado el sistema.

La Especificación de Requisitos utiliza a menudo, una estructura como clave para el entendimiento; desdichadamente no todos los sistemas tienen una estructura lo que imposibilita el propósito de la Especificación de Requisitos.

Para clarificar la diferencia entre el Diseño y el requerimiento podemos trazar dos deliberaciones principales de la actividad referida a la captura de requerimientos y análisis donde se puedan identificar dos fases:

1. Obtener información a grandes rasgos y entender el rol del sistema se espera operar en el ambiente seguido siempre de una descripción detallada de los requerimientos desde el punto de vista del usuario, además de que la tasa informal debe estar de acuerdo con lo que se tiene que hacer.
2. La Especificación totalmente detallada de los Requerimientos y el acuerdo de lo que va a entregarse.

Principios e ideas básicas

Hay diferentes enfoques para la generación de requerimientos, cada enfoque implica diferentes necesidades en el tipo de información que se captura, el nivel de riesgo en la aplicación y el método más apropiado empleado para la colección, los más comunes son:

Contractual:
Es una Especificación completa y se desarrolla previa al desarrollo del sistema. Es una estrategia apropiada cuando por ejemplo se utiliza el modelo de desarrollo en cascada.

Evolutiva:
Es una Especificación parcial de requerimientos (corazón del sistema) será desarrollada previa al desarrollo inicial por lo que se requiere una total funcionalidad subsecuente al análisis durante el desarrollo. Un ejemplo es cuando se utiliza el modelo de desarrollo en espiral.

La elección de los enfoques va a estar en dependencia de los escenarios que se espera que siga el desarrollo, los más comunes son:

- Reemplazo del sistema existente con poca o ninguna modificación.
- La integración de dos o más sistemas existentes.
- Un sistema completamente nuevo con usuarios conocidos, identificables, tangibles.
- Un sistema completamente nuevo con usuarios no identificables en la actualidad.

Los últimos dos deberán ser ajustados para la estrategia evolutiva y el resto para la contractual, aunque el escenario que se espera es una buena guía para la información que el análisis de requerimiento busca clarificar.

La fase de captura de requerimiento y análisis de un desarrollo se centra en:

1. El proceso empresarial que va ha ser apoyado y los beneficios empresariales por alcanzar.
2. La comunidad usuaria quiénes serán los usuarios y cómo van a interactuar con el sistema.
3. Análisis de mercado.
4. Sistema existente y su documentación.
5. Prioridad de requerimiento.

Una selección apropiada del método de análisis de requerimiento depende del enfoque general que debe relacionarse con el escenario donde influye el balance de la información requerida.

La información que proporciona la organización del comprador y vendedor solo será necesaria para comentar, corregir y cuestionar sobre el diseño y factibilidad; el problema es cuando el comprador no posea la habilidad de informar sobre todos los requerimientos, por la tanto es responsabilidad del proveedor reunir los requerimientos consistentes y realistas, para esto hay tres clases de preguntas claves:

1. Preguntas iniciales de los usuarios.
2. Cambios en los requerimientos originados por cambios en el ambiente del sistema.
3. Cambios en los requerimientos originados por el proceso de desarrollo.

El balance en las clases de preguntas anteriores depende del enfoque seleccionado y la etapa alcanzada en el proyecto, el proceso de captura de requerimiento y análisis debe corresponder a los siguientes objetivos:

- Agrupar información.
- Producir terminología adecuada.
- Verificación de consistencia interna y totalidad.
- Asegurar que la especificación de requerimientos refleja la capacidad de tecnología.
- Hacer una clara distinción entre requerimiento y el diseño.
- Identificar las limitaciones del sistema.
- Abarcar todas las vistas relevantes.
- Habilitar cambios controlados para introducirlos.
- Anotar las restricciones relevantes e impuestos.
- Asegurar que los requerimientos de calidad externa estén registrados de manera explícita.

Áreas del Problema

La preparación de una especificación de requerimientos es difícil, se debe de describir cada aspecto del comportamiento del sistema, de modo que se debe entender bien los problemas, deseos y ambiente del comprador, por lo tanto, el alcance de la especificaron de requerimientos hace de su producción una tarea especializada que a veces se hace más difícil de hacer por ciertos factores:

1. Tendencia a omitir la información que es considerada como obvia para el comprador.
2. Tanto los compradores como los proveedores utilizan terminología diferente provocando problemas de comunicaron.
3. Los compradores con frecuencia no están seguros acerca de las capacidades y limitaciones de la tecnología actual indicando requerimientos fuera de la serie.
4. La definición de los límites del sistema por lo general no se especifica con claridad.
5. Los diferentes usuarios tienen requerimientos del sistema no compatibles por lo que se hace necesario una reconciliación.
6. Los requerimientos siempre están sujetos a cambios.
7. La prueba de los requerimientos es por lo general imposible.
8. Es frecuente que se haga un compromiso debido a factores económicos de ingeniería y aceptabilidad.
9. La distinción entre los requerimientos del diseño se obstaculizan.

Estos factores y otros hacen que sea muy importante el contar con un buen método que tenga herramientas de soporte para el apoyo a la etapa del proceso de captura de requerimientos.

La elección principal que podemos atender consiste en que es vital saber los que el cliente de verdad desea.

4.1.1 Enfoques Actuales Para la Captura y Análisis de Requerimiento

***Requerimiento*:** Es lo que el sistema debe de hacer.
Para la captura y análisis de requerimiento hay muchas técnicas disponibles que resultan en la elaboración de la especificación de requerimientos.

Existen 3 elementos para capturar y analizar los requerimientos:

1. Estándares: define el contenido y criterios de los requerimientos especificados.
2. Métodos: son un conjunto de reglas para la adquisición análisis y prueba de la información.
3. Herramientas: son auxiliares automatizados que generalmente soportan el control del volumen de la información.

Estándares:

Los estándares existentes coinciden que para una especificación ideal de requerimiento debe tener las características siguientes:

Sin ambigüedad: lo que significa que cada requerimiento debe tener una sola interpretación para ayudarnos podemos usar lenguajes formales y semiformales.

- Completos: consiste en que debe incluirse tanto los requerimientos como los aspectos de su ambiente que se relacionan con la funcionabilidad desempeño y restricciones de diseño.
- Verificables: significa que el requerimiento se establezca de forma que el sistema pueda probarse para asegurar que el requerimiento ha sido satisfecho.
- Consistente: que no existe conflicto entre los requerimientos individuales.
- Modificables: los requerimientos deben organizarse de manera que permitan adaptares a las modificaciones que los usuarios hagan.
- Inteligibles: dado que los requerimientos tienen que ser leídos y entendidos por los usuarios, analistas, proveedores, compradores de manera que forman la base para la comunicación.

Métodos :

Los métodos de requerimientos caen en 3 categorías:

- Informales : la notación utilizada es por lo general lenguaje natural soportados por diagramas sin un significado formal ni predefinido, de manera que utilizan una notación sin sintaxis ni semántica fija.
- Semiformales: este método emplea notaciones gráficas con sintaxis definidas y semánticas bien establecidas también tienen un conjunto bien estructurado de

procedimientos y guías ofreciendo consistencia, totalidad, claridad, y son con frecuencia vehículos de comunicación.

- Formales: estos enfoques utilizan una notación matemática precisa que puede ser altamente expresiva no obstante tienden a una preparación alta de requerimiento y aun a facilitar su lectura y son de escasa concentración. Los costos de preparación para su desarrollo son aun más altos.

La mayoría de los métodos de captura y análisis de requerimientos utilizan un enfoque orientados a procesos o representaciones para expresar los requerimientos y proporcionar u marco para el análisis. Estas 2 versiones tienden hacia la visión de un sistema de ingieren.

Los enfoques orientados a representaciones se caracterizan por el modelo entidad relaciones y atributos (ERA) que permiten la descripción de los sistemas en términos de objetos de varios tipos de relaciones entre esas entidades sus propiedades especificadas.

Los enfoque orientados al proceso se caracterizan por los diagramas de flujos de datos que describe un sistema como una red de procesos conectados unos a otros por trayectorias mediante las cuales los datos pueden fluir siendo sus elementos básicos de estas redes procesos de flujos de datos y almacenamiento. Los DFD son jerárquicos cada proceso puede ser ampliado y detallado por varios procesos más.

MÉTODO CORE. Control de requerimientos y expresiones.

Es un conjunto de notaciones textuales y gráficas con guías especificadas para la captura y validación de requerimientos del sistema.

Este sistema soporta algunos aspectos de diseño tales como estructuras de datos.

Este método consiste de 4 etapas cada una produce salidas que aumentan a la etapa subsecuente como entrada o que forman parte de la especificación de requisitos final.

El método Core pretende examinar el sistema y sus ambientes en un número de niveles con detalles más finos en cada nivel.

Las etapas son:

1. Definición del problema: para identificar los limites del mismo y cuya salida es un documento de establecimiento del problema.
2. Estructuración del punto de vista: su propósito es descomponer el ambiente del sistema en elementos para que el sistema propuesto pueda ser analizado por los usuarios desde su punto de vista.
3. Colección tabular: en esta etapa se reúne la información sobre los flujos de datos.
4. Estructuración de datos: se da una vista más cercana al contenido de la estructura y a la desviación de datos al producir diagrama de estructuras de datos.

El área de aplicación del método Core ha sido en áreas de tiempo real como la industria de la aviación.

Herramientas:

La mayoría de las herramientas utilizan un lenguaje de especificación de requisitos el cual es un medio de comunicación de información entre el cliente y el desarrollador.

Un lenguaje de especificación de requerimientos tiene que ser:

- Preciso para permitir un conjunto de estatus de especificaciones cuya consistencia y totalidad serán probadas.
- Suficientemente natural para permitir la comprensión humana.

Existen 2 categorías de lenguaje de especificación de requerimiento:

1. Lenguaje de terminología científica.
2. Lenguaje de terminología definible.

Estos han sido aplicados con éxito para ayudar a la producción de especificaciones de requerimiento de mayor calidad que aquellos logrados manualmente.

Las herramientas basadas en computadoras proporcionan una ayuda automatizada en la producción y mantenimiento de especificaciones de requerimientos, al utilizar lenguajes de especificación de requerimientos estos comparten las siguientes características:

- Un sistema de base de datos que conserve el conjunto de especificaciones actual en una forma de total referencia cruzada.
- Facilidades para manipular los contenidos de la base de datos con pruebas.
- Presentación de facilidades para permitir la extracción y presentación de información guardada en la base de datos de acuerdos con criterios predefinidos para usuarios proveedor.

Enfoque General
En vista a que no existe una forma general acordada para realizar el análisis de requerimiento, abordaremos un modelo simple para esta primera parte del proceso de desarrollo.

En este enfoque se requieren tres definiciones para asegurar que los requerimientos del sistema se analizan en forma minuciosa:

- Una especificación de la situación actual.
- Una especificación de la situación meta.
- Una especificación de cómo puede transformarse la situación actual a una situación meta.

Tipos de Análisis

En cualquier sistema complejo, se requiere de muchas opiniones y consideraciones, por tanto se recomienda realizar el análisis abordando cada uno de los siguientes tipos de análisis:

Análisis de relevancia:

Se construye una tabla de requerimientos y aquellos a los que afecta.

Puede originarse una prioridad a los requerimientos de acuerdo con su relevancia total asegurada, de igual manera puede asegurarse un orden de prioridad de acuerdo con el numero y extensión de los requerimientos que lo afectan.

Análisis de Calidad:
Cualquier conjunto de requerimientos puede examinarse de acuerdo a un conjunto básico de atributos de calidad como son la eficiencia y la confiabilidad.

Análisis de Intuición:
Los requerimientos rara vez son independientes y podemos utilizar una tabla para clasificar la relación que se presenta y la estimación hechas de la dependencia involucrada, este tipo de análisis da un significado de requerimientos identificados que pueden ser seguidos por separado o bien agrupar aquellos que necesitan ser tratados en forma colectiva.

Análisis de Validación:
Aún cuando los requerimientos sean comprendidos con claridad deben de ser escritos y refinados, la técnica de validación nos ayuda a codificar el análisis con la documentación con el propósito de generar documentación extra que permita una reunión más rigurosa. La función de esta documentación extra es asegurar la idea como los conceptos presentados en una descripción.

La validación llama a una descripción a generarse en dos parte separadas pero relacionadas. Es un documento narrativo el cual contendrá varias palabras y frases que poseen significado preciso y particular. Es una referencia que proporciona una definición para todas las palabras claves y frases en la parte narrativa.

Lista de requerimientos
Dada la naturaleza de los requerimientos, es fácil perder atributos importantes que el sistema debe poseer. La única manera para minimizar todas las omisiones es asegurar que gente con experiencia confronte los requerimientos del sistema.

Dado que esto no es siempre posible, la mejor forma es emplear una lista.

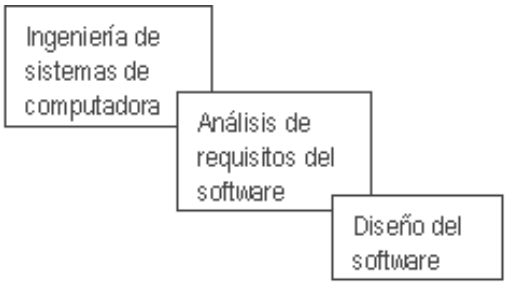
Existen 4 aspectos claves en la lista para la captura y análisis de requerimientos :

1. Información: la información reunida debe incluir los atributos de los usuarios (seguridad, desempeño, utilidad, confiabilidad, etc.), además de tomar en cuenta los afectados por el sistema.
2. Análisis: aquí debemos tomar en cuenta ciertos criterios para aquellos requerimientos prioritarios.
3. Aseguramiento: el cliente debe estar involucrado y comprometido a revisar el establecimiento de requerimientos.
4. Proceso de requerimiento: aquí se debe estar claro quien es el cliente que lo que necesita el sistema y las razones para las decisiones tomadas.

La lista abarca la mayoría de factores en la generación de un conjunto completo y bien estructurado de requerimientos del sistema.

Análisis de Requisitos

El Análisis de Requisitos es la tarea de la Ingeniería del Software que establece un puente entre la asignación del software a nivel de Sistema y el Diseño del Software



El Análisis de Requisitos facilita al Ingeniero de Sistemas la especificación de la función y del rendimiento del Software, la descripción de la interfaz con otros elementos del Sistema y el establecimiento de las restricciones de diseño que debe considerar el Software. El Análisis de Requisitos permite al Ingeniero de Software (frecuentemente llamado analista en este papel) refinar la asignación del Software y construir modelos de los ámbitos del proceso, de los datos y del comportamiento que serán cubiertos por el Software.

Por último, la Especificación de los Requisitos suministra al técnico y al cliente un medio para valorar la calidad del software una vez que se haya construido.

Tareas de Análisis

En el Análisis de los Requisitos del Software se pueden identificar cinco áreas de esfuerzo:

1. Reconocimiento del Problema.
2. Evaluación y Síntesis.
3. Modelización.
4. Especificación.
5. Revisión.

Se debe establecer una comunicación adecuada para el análisis, de forma que se facilite el reconocimiento del problema. El analista debe establecer contacto con el equipo técnico y de gestión del usuario/cliente y de la organización de desarrollo del Software. El objetivo del Analista es reconocer los elementos básicos del problema tal como lo percibe el usuario/cliente.

La evaluación del problema y la síntesis de la solución es la siguiente área principal del esfuerzo de análisis. El analista debe evaluar el flujo y la estructura de la información, definir y elaborar todas las funciones del Software, entender el comportamiento del programa en el contexto de los sucesos que afectan al Sistema, establecer las características de la interfaz del Sistema y descubrir las restricciones de Diseño. Cada una de éstas tares sirve para describir el problema de forma que pueda sintetizarse un enfoque o solución global.

A lo largo de la Evaluación y Síntesis de la Solución, el analista se centra básicamente en el "que", no en el "como". ¿Qué datos produce y consume el Sistema?, ¿qué funciones debe realizar el sistema?, ¿qué interfaces están definidas? y ¿qué restricciones se aplican?

Las tareas asociadas con el análisis y la especificación intentan proporcionar una representación del Software que pueda ser revisada y aprobada por el cliente.

Una vez que se han descrito la información básica, las funciones, el rendimiento, el comportamiento y la interfaz, se especifican los criterios de validación que han de servir para demostrar que se ha llegado ha un buen entendimiento de la forma de implementar con éxito el Software. Estos criterios sirven como base para las actividades de prueba que se llevarán a cabo más adelante dentro del proceso de la Ingeniería del Software. Para definir las características y los atributos del Software, se escribe una especificación formal de requisitos.

Los documentos del Análisis de Requisitos (especificación y manual de usuario) sirven como base para una revisión por parte del cliente y del desarrollador.

La revisión de los requisitos implica casi siempre modificaciones en la función, en el rendimiento, en la representación de la información, en las restricciones o en los criterios de validación. Además, se vuelve a evaluar el plan del proyecto de Software, para determinar si las primeras estimaciones siguen siendo válidas después del conocimiento adicional obtenido durante el análisis.

El Analista

Una descripción práctica del trabajo de un Analista: "…se espera que el Analista del Sistema analice y diseñe Sistemas con un rendimiento óptimo. Es decir el Analista debe producir… una salida que satisfaga completamente los objetivos de la gestión…".

Al Analista se le conoce con distintos nombres: Analista de Sistemas, Ingeniero de Sistemas, Diseñador, Jefe de Sistemas, Programador/Analista, etc. independientemente del título de su trabajo, el Analista debe exhibir los siguientes rasgos de carácter:

- Habilidad para comprender conceptos abstractos, reorganizarlos en divisiones lógicas y sintetizar "soluciones" basadas en cada división.
- Habilidad para entresacar hechos importantes de fuentes conflictivas o confusas.
- Habilidad para comprender entornos de usuario/cliente.
- Habilidad para aplicar elementos del Sistema de Hardware y/o de Software a entornos de usuario/cliente.
- Habilidad para comunicarse bien de forma escrita y verbal.
- Habilidad para evitar que "los árboles no lo dejen ver el bosque".

Probablemente, ésta última habilidad sea la que de verdad distingue a los buenos analistas. Los individuos que se paran excesivamente en los detalles, con frecuencia pierden de vista el objetivo global del Software. Se deben descubrir los requisitos del Software de una manera "descendente" –se deben comprender perfectamente las

funciones principales, las interfaces y la información antes de especificar los niveles suficientes de detalle.

Técnicas de comunicación

El analista de requisitos del software siempre comienza con una comunicación entre dos o más partes. Un cliente tiene un problema al que puede encontrar una solución basada en computadora. El desarrollador responde a la petición del cliente. La comunicación ha comenzado. Pero como ya hemos visto, el camino entre la comunicación y en entendimiento está lleno de baches.

La técnica más común es dirigir una entrevista o revisión preliminar donde se plantean preguntas al cliente para: determinar los objetivos generales y los beneficios que el cliente espera; para conocer mejor el problema y para determinar la efectividad de la reunión.

Principio de análisis

Cada método de análisis tiene una notación y punto de vista único. Sin embargo, todos los métodos de análisis están relacionados por un conjunto de principio fundamentales:

- Se debe representar y comprender el ámbito de información del problema.
- Se deben desarrollar los modelos que representen la información, función y el comportamiento del sistema.
- Se deben subdividir los modelos (y el problema) de forma que se descubran los detalles de una madera progresivo (o jerárquica)
- El proceso de análisis debe ir de la información esencial hacia el detalle de la implementación.

El ámbito de información se examina para poder comprender completamente la función. Los modelos se utilizan para poder comunicar la información de forma compacta. La partición se aplica para reducir la complejidad. Los planteamientos esenciales y de implementación del software son necesarios para acomodar las ligaduras lógicas impuestas por los requisitos de procesamientos y las ligaduras físicas impuestas por los elementos del sistema.

El ámbito de información

Todas las aplicaciones del software pueden colectivamente llamarse procesamientos de datos. Este término contiene la clave para la compresión de los requisitos del software. El software se construye para procesar datos: para trasformar datos de una forma a otra, es decir, para aceptar una entrada manipularla de alguna forma y producir una salida.

Sin embargo es importante tener en cuenta que el Software también procesa sucesos. Un suceso representa algún aspecto de control del sistema y realmente no es más que un dato Booleano.

El ámbito de información contiene tres planteamientos diferentes de los datos y del control a medida que son procesados por un programa de computadoras:

- El flujo de información.
- El contenido de información.
- La estructura de información.

El flujo de información representa la manera en la que los datos y el control cambian conforme se muevan a través de un sistema.

La entrada se transforma en información intermedia que es nuevamente transformada en la salida. A lo largo de este camino de transformación, puede introducirse información adicional procedente de un almacén de datos existentes (p. ejemplo: un archivo de disco o un buffer de memoria). Las transformaciones que se aplican a los datos son funciones o subfunciones que un programa debe ejecutar. El control y los datos que se mueven entre dos transformaciones (funciones) definen la interfaz de cada función.

El contenido de información representa los elementos de datos individuales que componen otros elementos mayores de información. Por ejemplo, el elemento nómina es una composición de varias piezas importantes de información: el nombre del empleado, el sueldo neto, el sueldo bruto, las deducciones, etc. Por ello, se define el contenido de nómina con los elementos que se necesitan para crearlo.

La estructura de información representa la organización interna de los distintos elementos de datos y de control. Dentro del contexto de la estructura, ¿qué información esta relacionada con otra información? ¿Esta toda la información contenida dentro de una estructura simple de información o se utilizan distintas estructuras?. Estas preguntas y otras son respondidas al establecer la estructura de información.

Modelización
Cuando la entidad a construir es Software, nuestro modelo debe de ser capaz de modelizar la información que transforma el Software, las funciones y subfunciones que permiten que se produzcan la transformación y el comportamiento del sistema a medida que se produce la transformación.

Durante el análisis de los requisitos del Software, creamos modelos del sistema a construir. Los modelos se centran en lo que tiene que hacer el sistema y no en como lo tiene que hacer.

Los modelos creados durante el análisis de requisitos desempeñan varios papeles importantes:

- El modelo ayuda al analista a entender la información, la función y el comportamiento del sistema, haciendo por ello que la tarea de análisis de requisitos sea más fácil y más sistemática.
- El modelo se convierte en el punto focal para la revisión y por tanto, en la clave para la determinación de la integridad, la consistencia y la eficacia de la especificación.
- El modelo se convierte en la base del diseño, proporcionando al diseñador una representación esencial del Software que se puede hacer corresponder con un contexto de implementación.

Partición

A menudo, los problemas son demasiado grandes y complejos para que se puedan comprender como un todo. Por esta razón, tendemos a partir dichos problemas en partes que se puedan entender fácilmente y establecer interfaces entre las partes, de forma que se realice la función global.

En esencia la partición descompone un problema en sus partes constituyentes. Conceptualmente, establecemos una representación jerárquica de la función o de la información y luego descomponemos el elemento superior de la siguiente forma:

1. Exponiendo mas detalles, al movernos verticalmente por la jerarquía.
2. Descomponiendo funcionalmente el problema, al movernos horizontalmente por la jerarquía.

Planteamientos esenciales y de implementación

El planteamiento esencial de los requisitos del software presenta las funciones que han de realizarse y la información que ha de procesarse, independientemente de los detalles de implementación. Por ejemplo, el planteamiento esencial de la función lectura del estado del sensor de hogar seguro no se refiere en sí a la forma física de los datos o al tipo de sensor usado. De hecho, podría argumentarse que lectura de estado es un nombre más apropiado para esa función, puesto que no consideran en absoluto los detalles sobre el mecanismo de entrada.

El planteamiento de implementación de los requisitos del software presenta la manifestación en el mundo real de las funciones de procesamientos y de las estructuras de información. En algunos casos, la representación física se desarrolla en el primer paso del diseño del software. Sin embargo, la mayoría de los sistemas basados en computadoras se especifican de alguna forma teniendo en cuenta ciertos detalles de implementación. El dispositivo de entrada de Hogarseguro es un sensor de perímetro (no es un perro guardián, ni un guardia de seguridad, ni una trampa). El sensor detecta una entrada ilegal mediante una interrupción de un circuito electrónico. Se deben contemplar las características generales del sensor como parte de la especificación de requisito del software.

Ya hemos visto que el análisis de requisitos del software debe enfocarse sobre qué debe realizar el software, en vez de cómo se va a implementar el procesamiento. Sin embargo, planteamiento físico no debe ser interpretado necesariamente como una representación del cómo. En cambio, el modelo de implementación representa el modo real de operación, es decir la asignación existente o propuesta para todos los elementos del sistema.

4.2 TIPOS DE REQUERIMIENTOS

Requerimientos funcionales.- especifican los servicios que debe proporcionar la aplicación (por ejemplo, “la aplicación debe calcular el valor del portafolio de inversión del usuario”). Por otro lado, un requerimiento como “la aplicación debe terminar el cálculo del valor de cada portafolio en menos de un segundo” no es un requerimiento funcional, porque no especifica un servicio dado. En su lugar, califica un servicio (especifica algo sobre ellos).

Requerimientos no funcionales: requerimiento de desempeño.- especifican las restricciones de tiempo que debe observar la aplicación. Los clientes y desarrolladores negocian las restricciones del tiempo transcurrido para los cálculos, el uso de RAM, el almacenamiento secundario, etc. Por ejemplo: Para cualquier viga, el analizador de tensión debe producir un informe del tipo cinco en menos de un minuto

Requerimientos no funcionales: confiabilidad y disponibilidad.- especifican la confiabilidad en términos cuantificados. Este tipo de requerimiento reconoce que es poco probable que las aplicaciones sean perfectas, por lo que circunscribe su grado de imperfección. Por ejemplo: La aplicación de radar del aeropuerto (ARA) debe experimentar no más de dos fallas de nivel uno por mes.

La disponibilidad, que tiene una estrecha relación con la confiabilidad, cuantifica el grado en el que la aplicación debe estar disponible para los usuarios

Requerimientos no funcionales: manejo de errores.- explica cómo debe responder la aplicación a los errores en su entorno. Por ejemplo, ¿qué debe hacer la aplicación si recibe un mensaje de otra que no está en el formato que se acordó? Estos no son errores generados por la aplicación.

En algunos casos, el “manejo de errores” se refiere a las acciones que debe realizar la aplicación si encuentra que ella misma cometió un error debido a un defecto en su construcción.

Sin embargo, este tipo de requerimientos de errores debe aplicarse en forma selectiva porque nuestro objetivo es producir aplicaciones libres de defectos, y no cubrir los errores con códigos interminables de manejo de errores

Requerimientos no funcionales: requerimiento de interfaz.- describen el formato con el que la aplicación se comunica con su entorno. Por ejemplo: El costo de enviar el artículo de una fuente a un destino debe desplegarse en todo momento en el cuadro de “costo”

Requerimientos no funcionales: restricciones.- las restricciones de diseño o implementación describen los límites o condiciones para diseñar o implementar la aplicación. Estos requerimientos no pretenden sustituir el proceso de diseño, sólo especifican las condiciones que el cliente impone al proyecto, el entorno u otras circunstancias, e incluyen la exactitud, por ejemplo: Los cálculos de daños de la Automobile Impact Factility (AEF) deben tener una exactitud de un centímetro

Requerimientos inversos.- establecen qué no debe hacer el software. Es lógico que haya un número infinito de requerimientos inversos: se selecciona los que aclaran los requerimientos verdaderos y los que eliminan posibles malos entendidos. Por ejemplo: No se requiere que AEF analice datos de choque.

4.3 ANÁLISIS DEL SISTEMA

El análisis del sistema es una actividad que engloba la mayoría de las tareas que hemos llamado colectivamente ingeniería de sistemas basados en computadora.

El análisis del sistema se realiza teniendo presente los siguientes objetivos:

1. identificar las necesidades del cliente
   - entrevistas con el cliente
   - objetivos del sistema
   - funcionamiento y rendimiento requeridos
   - aspectos de fiabilidad y calidad
   - limitaciones de coste/agenda
   - requisitos de fabricación
   - ampliaciones futuras
2. evaluar la viabilidad del sistema
   - económica
   - corto plazo
   - costo-beneficio
   - largo plazo
   - técnica
   - riesgo del desarrollo
   - disponibilidad de recursos
   - tecnología
   - legal
   - contratos
   - derechos
   - alternativas
   - tiempo
   - dinero

3a realizar un análisis económico
   - justificación económica
   - análisis coste-beneficio

3b realizar un análisis técnico
   - rendimiento
   - fiabilidad
   - mantenimiento
   - producción

4 asignar funciones al software, al hardware, a la gente, a la base de datos y a otros elementos del sistema

5 establecer relaciones de coste y tiempo

6 crear una definición del sistema que sea base para todo el trabajo posterior de ingeniería

Modelización de la arquitectura del sistema

El ingeniero de sistemas debe crear un modelo que represente los límites de la información y las interrelaciones entre los distintos elementos del sistema y establezca una base para los posteriores pasos de análisis de requisitos y diseño.

1. Diagramas de arquitecturas

Para desarrollar el modelo del sistema se utiliza una plantilla de arquitectura definiendo:

- Procesamiento de la interfaz del usuario
- Procesamiento de la entrada
- Funciones de proceso y control del sistema

- Procesamiento de la salida
- Mantenimiento y autocomprobación

2. Especificación de la arquitectura del sistema

Se puede especificar los subsistemas y la información que fluye entre ellos para que esté disponible en los posteriores trabajos de ingeniería. La especificación del diagrama de arquitectura presenta la información sobre cada subsistema y el flujo de información entre los subsistemas.

Simulación y modelización del sistema

Muchos sistemas basados en computadoras interactúan con el mundo real de una forma reactiva. Si el sistema falla pueden producirse pérdidas humanas o económicas importantes. Hoy en día se usan herramientas CASE (computer assisted software engineering = ingeniería del software asistida por computadoras) para la modelización y la simulación, con el fin de eliminar sorpresas en la construcción de sistemas reactivos basados en computadora. Estas herramientas se aplican durante el proceso de ingeniería del software, durante la especificación de los papeles del hardware, del software, de la gente y de las bases de datos.

Las herramientas de modelización y simulación permiten al ingeniero de sistemas "conducir la prueba" de una especificación del sistema.

La especificación del sistema
La especificación del sistema es un documento que sirve como base para la ingeniería del hardware, la ingeniería del software, la ingeniería de bases de datos y la ingeniería humana. Describe la función y el rendimiento de un sistema basado en computadora y las restricciones que gobernarán su desarrollo.

Revisión de la especificación del sistema
La revisión de la especificación del sistema evalúa la corrección de la definición contenida en la especificación del sistema. La revisión ha de ser realizada por el técnico y por el cliente, para asegurar que:

- se ha perfilado adecuadamente el ámbito del proyecto;
- se ha definido correctamente la funcionalidad, el rendimiento, y las interfaces;
- el análisis del entorno y del riesgo del desarrollo justifican el sistema; y
- el desarrollador y el cliente tienen la misma percepción de los objetivos del sistema.

Se realiza en dos partes:
1. Se aplica un punto de vista de gestión
2. Se realiza una evaluación técnica de los elementos y funciones del sistema.

La especificación, independiente del modo en que se realice, puede ser vista como un proceso de representación. Los requisitos se representan de forman que conduzcan finalmente a una correcta implementación del software. *BALZER Y GOLDMAN* proponen ocho principios para una buena especificación los cuales son:

PRINCIPIO # 1. Separar funcionalidad de implementación.

En primer lugar, una especificación, por definición, es una descripción de lo que se desea realizar no de cómo se va a realizar (implementar).

PRINCIPIO # 2. Se necesita un lenguaje de especificación de sistemas orientado al proceso.

Aquí debe emplearse una descripción orientada al proceso en la cual la especificación del qué se consigue mediante la especificación de un modelo del comportamiento deseados en términos de respuestas funcionales a distintos estímulos del entorno.

PRINCIPIO # 3. Una especificación debe abarcar el sistema del cual el software es un componente.

Un sistema está compuesto de componentes que interactúan. Sólo dentro del contexto del sistema completo y la interacción entre sus partes puede ser definido el comportamiento de un componente específico.

PRINCIPIO # 4. Una especificación debe abarcar el entorno en el que el sistema opera.

Similarmente, debe especificarse el entorno en el que opera el sistema y con el que interactúa.

La única diferencia que existe entre el sistema y su entorno es que el esfuerzo de diseño y de implementación posterior operará exclusivamente sobre la especificación del sistema. La especificación del entorno permite especificar la “interfaz” del sistema de la misma forma que el propio sistema, en vez de introducir otro formalismo.

PRINCIPIO # 5. Una especificación de sistema debe ser un modelo cognitivo.

La especificación de un sistema debe ser un modelo cognitivo en vez de un modelo de diseño o de implementación. Debe de describir un sistema tal como es percibido por su comunidad de usuarios.

PRINCIPIO # 6. Una especificación debe ser operativa.

La especificación debe ser completa y lo bastante formal para que pueda usarse para determinar si una implementación propuesta satisface la especificación con casos de prueba elegidos arbitrariamente. Esto implica que la especificación, sin ser una especificación completa del como, puede actuar como un generador de posibles comportamientos, entre los que debe estar la implementación propuesta. Por tanto, en un sentido amplio, la especificación debe ser operativa.

PRINCIPIO # 7. La especificación del sistema debe ser tolerante a la incompletitud y ampliable.

Ninguna especificación puede ser totalmente completa. El entorno en el que existe es demasiado complejo para que lo sea. Una especificación es siempre un modelo de alguna situación real o imaginada. Por tanto, será incompleta. Las herramientas de análisis empleadas para ayudar a los especificadores y para probar las especificaciones. Deben ser capaces de tratar con la incompletitud.

PRINCIPIO # 8. Una especificación debe estar localizada y débilmente acoplada.

Los principios anteriores tratan la especificación como una entidad estática. Este surge de la dinámica de la especificación. Debe reconocerse que aunque el principal propósito de una especificación sea servir como base para el diseño y la implementación de algún sistema, no es un objeto estático previamente compuesto, sino un objeto dinámico que sufre considerables modificaciones. Tales modificaciones aparecen en tres actividades principales que son:

- **FORMULACIÓN:** Cuando se esta creando una especificación inicial.
- **DESARROLLO:** Cuando se esta elaborando la especificación con el proceso iterativo de diseño o implementación.
- **MANTENIMIENTO:** Cuando se cambia la especificación para que refleje los cambios en el entorno y/o requisitos funcionales adicionales

4.3.1 ANÁLISIS ESTRUCTURADO

El análisis estructurado se basa en el modelo del flujo como primer elemento de la representación gráfica de un sistema basado en computadora. Usando como base diagramas de flujo de datos y de control, el analista separa las funciones que transforman el flujo. Después, crea un modelo de comportamiento usando el diagrama de transición de estados y un modelo de contenido de los datos con un diccionario de requisitos. Las especificaciones de los procesos y del control proporcionan una elaboración adicional de los detalles.

Mecánica del análisis estructurado

Introducción

El objetivo que persigue el análisis estructurado es organizar las tareas asociadas con la determinación de requerimientos para obtener la comprensión completa y exacta de una situación dada.

Para poder utilizar eficientemente las notaciones básicas en el análisis de requisitos del Software, se ha de combinar esa notación con un conjunto de heurística que permitan al Ingeniero del Software derivar un buen modelo de análisis.

El análisis estructurado nos permite:

1. Intentar estructurar el proceso de determinación de los requerimientos.
2. Incluir todos los detalles relevantes que describen al sistema en uso.
3. Una fácil verificación cuando se han omitido detalles relevantes.
4. La identificación de los requerimientos.
5. Generar una documentación más eficiente.

A continuación se examina cada uno de los pasos que se deben seguir para desarrollar modelos completos y precisos mediante el análisis estructurado.

Creación de un modelo de Flujo de Datos

A medida que se refina el DFD en mayores niveles de detalle, el analista lleva a cabo implícitamente una descomposición funcional del sistema. Al mismo tiempo, el refinamiento del DFD produce un refinamiento de los datos a medida que se mueven a través de los procesos que componen la aplicación.

Unas pocas directrices sencillas pueden ayudar de forma considerable durante la derivación de un diagrama de flujo de datos:

- El diagrama de flujo de datos de nivel 0 debe reflejar el software / sistema como una sola burbuja.
- Se deben anotar cuidadosamente la entrada y la salida principales.
- El refinamiento debe comenzar aislando los procesos, los elementos de datos y los almacenes de datos que sean candidatos a ser representados en el siguiente nivel.
- Todas las flechas y las burbujas deben ser etiquetadas con nombres significativos.
- Entre sucesivos niveles se debe mantener la *continuidad del flujo de información.*
- Se deben refinar las burbujas de una en una.

Creación de un modelo de flujo de control

Para muchos tipos de aplicaciones de procesamiento de datos, todo lo que se necesita para obtener una representación significativa de los requisitos del Software es el modelo de flujo de datos. Sin embargo existe una clase de numerosas aplicaciones que están “conducidas” por sucesos en lugar de por los datos, que producen información de control más que informes, que procesan información con fuertes limitaciones de tiempo y rendimiento. Tales aplicaciones requieren una modelización del flujo de control además de la modelización del flujo de la información.

La especificación de control

Esta representa el comportamiento del sistema de dos formas diferentes. La especificación de control contiene un diagrama de transición de estados (DTE) que es una especificación secuencial del comportamiento. También puede contener una tabla de activación de procesos (una especificación combinatoria del comportamiento).

La especificación de procesamiento

Se usa para describir todos los procesos del modelo de flujo que aparece en el nivel final del refinamiento. El contenido de la especificación de procesamiento puede incluir una narrativa textual, una descripción en lenguaje de descripción de programa del algoritmo del proceso, tablas, diagramas o gráficos.

El diccionario de requisitos

El diccionario de requisitos también denominado diccionario de datos, es una gramática casi formal para describir el contenido de los objetos definidos durante el análisis estructurado.

El diccionario de datos es una listado organizado de todos los elementos de datos que son pertinentes para el sistema, con definiciones precisas y rigurosas que permiten que el usuario y el analista del sistema tengan una misma comprensión de las entradas, las salidas, de las componentes de los almacenes y de los cálculos intermedios.

Un diccionario de datos contiene la siguiente información:

1. **Elementos de datos:** Campo-dato-atributo: Es la menor unidad de información con significado para el analista.
2. **Estructuras de datos:** Grupo de datos elementales que se relacionan con otros y en conjunto constituyen un componente del sistema.

Descripción de los elementos dato

- **Nombre de los datos**: comprensible y significativo. se provee para distinguir un dato de otro.
- **Descripción de los datos:** que representa el dato para el sistema. Se supone que el lector desconoce el sistema.
- **Alias:** diferentes nombres con los que se conoce el dato.
- **Longitud:** Cantidad de espacios necesarios para almacenarlo.
- **Valores de los datos:** Conjunto específico de valores permitidos a los datos. Validación - Valores por defecto.

Descripción de las estructuras de datos

Relación secuencial: Incluye un conjunto definido de componentes. Los elementos están incluidos en la estructura de datos. Ej:

NOMBRE :
    Nombre
    Primer apellido
    Segundo apellido
DIRECCIÓN
CIUDAD
DEPARTAMENTO
APARTADO AÉREO
NÚMERO _ TELEFÓNICO

Relación de selección: Define alternativas para datos o estructuras de datos incluidas en una estructura de datos. Debe seleccionarse el objeto de un conjunto de dos o más objetos. Ej:

ESTUDIANTE
NOMBRE
DIRECCIÓN
CIUDAD
ESTADO
APARTADO AÉREO
NUMERO TELEFÓNICO
Y uno de los siguientes: TARJETA DE IDENTIDAD
NRO DOCUMENTO DE IDENTIDAD
CEDULA DE EXTRANJERÍA

Relación de iteración: se presenta cuando los elemento que componen una estructura de datos están repetidos. Ej;

MATRICULA DEL SEMESTRE:
PERIODO
AÑO
Desde una hasta seis iteraciones del CURSO:
CÓDIGO DE LA MATERIA
NOMBRE DEL MATERIA
NUMERO DE CRÉDITOS
DEPARTAMENTO QUE OFRECE
HORARIO
DÍAS
PROFESOR

Relación Opcional: Algunos elementos dato pueden o no estar incluidos. Puede tomarse como iteración uno o cero. Representan datos opcionales. Ej:

- Los datos del cónyuge: nombre y apellido, nro telefónico, dirección, actividad que desempeña, etc.
- Los pagos opcionales en cierta universidad: Cuota por actividades deportivas, Recargo por pago extemporáneo, cuota de servicio médico, cuota por parqueadero, etc.

Notación Empleada

Debe emplearse una notación formal. Debe utilizarse símbolos sencillos. Los más empleados son:

=	está compuesto de
{}	Iteración
+	y
**	delimita comentarios
()	opcional (puede estar presente o ausente)
[]	Seleccionar una de varias alternativas (uno u otro)
|	Separa opciones alternativas en la construcción (o)

Ej:

nombre	= título de cortesía+nombre+(segundo nombre)+apellido
título de cortesía	= [Sr. | Sra. | Srta. | Dr. | Profesor]
nombre	= {carácter legal}
segundo nombre	= {carácter legal}
apellido	= {carácter legal}
carácter legal	= {A-Z | a-z |0-9 | ‘ | |}

FLUJOS DE DATOS

Cada flujo recibe un nombre y se describe de manera breve. Debe definirse de qué proceso, depósito o entidad externa proviene; a cual proceso, entidad externa o depósito se dirige; y la estructura de datos que maneja en forma general.

Ejemplo:

NOMBRE DEL FLUJO:	Solicitud de servicio
DESCRIPCIÓN:	Contiene detalles a cerca del pedido que un huésped ha realizado a cualquier dependencia del hotel (lavandería, teléfono, comedor, etc.)
PROVENIENTE DE:	Huéspedes
DESTINO:	Recepción Hotel
ESTRUCTURA DE DATOS:	Número del cuarto+tipo de servicio+valor+fechadel servicio+...

DEPÓSITOS

Cada depósito recibe un nombre y debe describirse de manera breve. Debe definirse qué flujos recibe y los flujos que proporciona; deben describirse los datos que maneja, el volumen o cantidad de datos que contiene y la forma como se accede a en forma general.

Ejemplo:

NOMBRE DEL DEPÓSITO:	Facturas
DESCRIPCIÓN:	Contiene datos sobre la factura que se le entrega a los clientes. FLUJOS DE DATOS RECIBIDOS: Factura-evento
FLUJOS DE DATOS ENTREGADOS :	Monto-factura-individual Factura
DESCRIPCIÓN DE DATOS:	Número de la factura+Fecha+Nombre del cliente+Detalle de la factura+Valor de la factura+...
VOLUMEN:	Aproximadamente 75 en semana, 120 en fin de semana hay mayor trabajo en puente
ACCESO:	Se procesa en lote, en forma secuencial.

PROCESOS

Cada proceso que se defina en un DFD debe definirse por separado en el DD.

Ejemplo:
NOMBRE DEL PROCESO: Salida Huéspedes
DESCRIPCIÓN: Procesa la información relacionada con la salida de los huéspedes: cuentas de cobro, informes del cuarto, liquidación de servicios, etc.
ENTRADAS: Consulta-servicios
datos-cuarto
pago
SALIDAS: Salida-huésped
cuenta-de-cobro
cierre-registro
informe-cuarto

LÓGICA DEL PROCESO: Se toman los datos sobre el servicio prestado, los datos del cuarto y se debe hacer un cierre en el libro de registro, elaborar una cuenta de cobro, consignar un informe del cuarto así como confirmar la salida del huésped.

OTROS DETALLES DEL DD

El DD debe contener definiciones concisas de los elementos del sistema. En el DD también se puede incluir:

1. Un listado de los elementos dato y la estructura de datos, incluyendo: nombre, descripción longitud y alias.
2. Listado de todos los procesos que se llevan a cabo en el sistema junto con una descripción de los mismos.
3. Verificación con referencias cruzadas: donde se emplean los datos en el sistema, que procesos los usan y que datos se emplean.
4. Detección de errores: Descubrimiento de inconsistencias, datos necesarios en un proceso que nunca ingresan, procesos que no son alimentados, procesos con flujos entrando y donde no se producen salidas.

El análisis estructurado es el método más utilizado en la modelización de requisitos, se basa en el modelo de flujo como primer elemento de representación gráfica de un sistema basado en computadoras. Usando como base diagramas de flujo de datos y de control, el analista separa las funciones que transforman el flujo. Después, crea un modelo de comportamiento usando el diagrama de transición de estados y un modelo de contenido de los datos con un diccionario de requisitos. Las especificaciones de los procesos y del control proporcionan una elaboración adicional de los detalles.

4.3.2 ANÁLISIS ORIENTADO A OBJETOS

A inicio de los años 80 el tema orientado a los objetos comienza a surgir en la ingeniería del software con un enfoque de desarrollo de software. Se empezó a diseñar e implementar aplicaciones usando la forma de pensar orientada a objetos.

Actualmente el análisis orientado a los objetos va progresando poco a poco como método de análisis de requisitos y como complemento de otros métodos de análisis. En lugar de examinar un problema mediante el método clásico entrada – salida o mediante un modelo derivado, el AOO introduce conceptos nuevos.

Conceptos orientados a los objetos

Clase: Es una descripción para producir objetos de esa clase o tipo. Una clase esta formada por los métodos y los atributos, que definen las características comunes a todos los objetos de esa clase. Una clase se puede considerar como una plantilla para crear objetos de esa clase o tipo.

Objetos: Es una encapsulación genérica de datos y de los procedimientos para manipularlos. Dicho de otra forma un objeto es una entidad que tiene unos atributos particulares (los datos) y una forma de operar sobre ellos (los métodos y los procedimientos).

Mensajes: Cuando se utiliza el AOO, los objetos están recibiendo, interpretando y respondiendo a mensajes de otros objetos. El conjunto de mensajes a los que un objeto puede responder se le llama protocolo.

Métodos: Un método determina cómo tiene que actuar el objeto cuando recibe un mensaje. Un método también puede enviar mensajes a otros objetos solicitando una acción o información. La descripción de un método se denomina operación.

Herencia: Es el mecanismo para compartir automáticamente métodos y datos entre clases y subclases.

Encapsulamiento: Se refiere a la práctica de incluir dentro de un objeto todo lo que necesita, de tal forma que ningún otro objeto necesite conocer nunca su estructura interna.

Polimorfismo: Es una característica que permite implementar múltiples formas de un mismo método, dependiendo de cada una de ellas de la case sobre la que se realice la implementación.

Identificación de objetos.

Cuando observamos el espacio del problema de una aplicación de software resulta complicado encontrar, a simple vista, los objetos.

Podemos empezar a identificar objetos examinando la descripción del problema. Si un objeto es necesario para implementar una solución, entonces es parte del espacio de la solución; en caso contrario cuando el objeto solo es necesario para describir la solución, entonces es parte del espacio del problema.

Los tipos de objetos pueden ser:

- Entidades externas: Son los que producen o consumen información a ser utilizadas en el sistema, por ejemplo otros sistemas, dispositivos, gente, etc.
- Cosas: Son parte del dominio de información del problema, por ejemplo, informes, visualizaciones, cartas, etc.

- Ocurrencias o sucesos: Es lo que ocurre en le contexto de operación del sistema, por ejemplo, cerrar una ventana en Windows, movimiento de un robot, etc.
- Papeles: Documento que juegan las personas que interactúan con el sistema, por ejemplo, Ingenieros, Vendedores, etc.
- Unidades organizativas: Las áreas que son relevantes para la aplicación, por ejemplo, división, grupo, equipo, departamento, etc.
- Lugares: Es el lugar que establece el contexto del problema y del funcionamiento general del sistema, por ejemplo, sala de facturación, muelle de descarga, etc.
- Estructuras: Aquellas que definen clases de objetos o en casos extremos clases de objetos relacionados, por ejemplo, sensores, computadoras, etc.

Especificación de atributos

Los atributos describen un objeto que haya sido seleccionado para ser incluido en el modelo de análisis. Esencialmente son estos los que definen un objeto; son los que aclaran el significado del objeto en el contexto del espacio del problema.

Para desarrollar un conjunto significativo de atributos para un objeto, el analista debe estudiar de nuevo la narrativa de procesamiento (o descripción del alcance) para el problema concreto y seleccionar aquello que pertenezca de forma razonable al objeto. Además se debe responder a la siguiente pregunta para cada objeto ¿Qué elementos de datos (elementales y/o compuesto) definen completamente ese objeto en el contexto del problema actual?

Operaciones básicas con objetos

Una operación cambia un objeto de alguna forma, correctamente cambia valores de uno o más atributos que están contenidos en el objeto. Por tanto una operación debe tener “conocimiento” de la naturaleza de los atributos del objeto y deben ser implementados de forma que se les permita manipular la estructura de datos que hayan sido derivados de los atributos.

Las operaciones generalmente se pueden dividir entre grandes categorías:

- Operaciones que manipulan los datos de alguna forma, por ejemplo adiciones, eliminaciones, cambio de formatos, selecciones, etc.
- Operaciones que realizan algún cálculo.
- Operaciones que monitorizan un objeto frente ocurrencia de algún suceso de control.

Comunicación entre objetos

La definición de los objetos del contexto del modelo de análisis puede ser suficiente para establecer una base para el diseño, pero se debe añadir algo más (durante el análisis o durante el diseño) para que se pueda construir el sistema se debe establecer un mecanismo para la comunicación entre los objetos; a este mecanismo se denomina mensaje.

La mensajería es importante para la implementación de un sistema orientado a los objetos; pero no es necesario considerarla detalladamente durante el análisis de requisitos. De hecho, en esta etapa nuestro único objetivo es usar el concepto como ayuda para la determinación de operaciones candidatas para un objeto concreto.

Fin de la definición del objeto

La definición de las operaciones es el último paso para tener la especificación de un objeto completo. Se puede determinar operaciones adicionales considerando el historial de un objeto y los mensajes que se pasan entre objetos definidos en le sistema.

El historial genérico de un objeto puede ser definido reconociendo que se debe crear, modificar, manipular o leer en otras formas y posiblemente eliminar una vez que se han definido los atributos y las operaciones para cada uno de los objetos hasta este nivel, se puede crear un modelo de AOO inicial.

Modelo de análisis orientado a los objetos

Se han sugerido varios esquemas de modelización para el AOO. Todos hacen uso de la definición de objeto, pero cada uno introduce sus propias notaciones, heurísticas y filosofía.

El enfoque de AOO propuesto por Coad y Yourdon consiste en cinco pasos:

1. identificación de los objetos
2. identificación de las estructuras
3. definición de temas
4. definición de conexiones entre atributos e instancias
5. definición de conexiones entre operaciones y mensajes

Estructuras de clasificación y ensamblaje

Una vez que se han identificado los objetos, el analista pasa a centrarse en la *estructura de clasificación* donde se considera cada objeto como una generalización y luego como una especialización, es decir, se definen y se nombran las instancias de un objeto. En otros casos, puede que un objeto representado en el modelo inicial este realmente compuesto por varias partes constitutivas que pueden ser definidas en si misma como objetos, a este tipo de estructura se denomina *estructura de ensamblaje*.

Definición de temas

Un tema no es más que una referencia o puntero a mayores detalles del modelo de análisis.

Al nivel más abstracto, el modelo de AOO contendrá sólo referencias temáticas.

Conexiones de instancias y caminos de mensajes

Una conexión de instancia es una notación de monetización que define una relación especifica entre las instancias de un objeto

Pasos para la creación de conexiones de instancias:

1. La creación de líneas sencillas entre los objetos indicando que existe alguna relación importante.
2. Una vez establecidas las líneas se evalúa cada extremo para determinar si lo que existe es una conexión simple (1:1) o una conexión múltiple (1:muchos). Para una conexión 1:1 la línea lleva una barra y para una conexión 1:muchos aparece con el símbolo de triple bifurcación.
3. Finalmente se añade otra barra vertical si la conexión es obligada y un circulo si la conexión es opcional

Al representar las conexiones de instancias el analista añade otra dimensión al modelo AOO. No sólo se han identificado las relaciones entre objetos sino que se han definido los caminos de mensajes importantes; en la figura que hace referencia a la definición de tema, las flechas punteadas que conectaban los símbolos temáticos se trataban de caminos de mensajes, cada flecha implica un intercambio de mensajes entre objetos del modelo. Los mensajes se moverán a través de los caminos de conexión de instancias, por tanto las conexiones de mensajes se derivan directamente de las conexiones de instancias.

AOO y Prototipos

El uso de AOO puede llevar a una creación de prototipos muy efectivos como a “técnicas de ingeniería del software evolutiva”.

Cuando se aplica AOO a nuevos proyectos, el analista puede trabajar en su especificación de sistema utilizando objetos existentes (implementados) que están contenidos en la biblioteca de objetos en lugar de inventar objetos nuevos.

Usando objetos existentes durante el análisis, el tiempo de especificación se reduce substancialmente, pudiéndose crear rápidamente un prototipo de sistema especificado que pueda ser revisado por el cliente.

El prototipo puede evolucionar hasta un producto o sistema operativo debido a que los bloques constitutivos que lo conforman (los objetos) ya han sido probados en la práctica.

Modelización de datos

La técnica de modelización de datos se centra únicamente en los datos, representando una “red de datos” existente para un sistema dado. La modelización de datos es útil para aplicaciones en las que los datos y las relaciones que gobiernan los datos son

complejas. A diferencia del enfoque estructurado, la modelización de datos considera los datos independientemente del procesamiento que transforma los datos.

La terminología de la modelización de datos y algunas de sus notaciones son similares a las usadas por el AOO. Pero es muy importante reconocer ambos enfoques como distintos, con un punto de vista bastante diferente.

La modelización de datos se usa ampliamente en aplicaciones de bases de datos. Proporciona al analista y al diseñador de la base de datos una amplia visión de los datos y las relaciones que gobiernan los datos.

Diagrama de Casos de Uso

Un diagrama de Casos de Uso muestra las distintas operaciones que se esperan de una aplicación o sistema y cómo se relaciona con su entorno (usuarios u otras aplicaciones).

Diagrama de Secuencia

Un diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo. Esta descripción es importante porque puede dar detalle a los casos de uso, aclarándolos al nivel de mensajes de los objetos existentes, como también muestra el uso de los mensajes de las clases diseñadas en el contexto de una operación.

Diagrama de Colaboración

Un diagrama de colaboración es una forma de representar interacción entre objetos, alterna al diagrama de secuencia. A diferencia de los diagramas de secuencia, pueden mostrar el contexto de la operación (cuáles objetos son atributos, cuáles temporales, ...) y ciclos en la ejecución.

Diagrama de Estados

Muestra el conjunto de estados por los cuales pasa un objeto durante su vida en una aplicación, junto con los cambios que permiten pasar de un estado a otro.

Diagrama de Actividades

Un diagrama de actividades es un caso especial de un diagrama de estados en el cual casi todos los estados son estados de acción (identifican que acción se ejecuta al estar en él) y casi todas las transiciones son enviadas al terminar la acción ejecutada en el estado anterior. Puede dar detalle a un caso de uso, un objeto o un mensaje en un objeto. Sirven para representar transiciones internas, sin hacer mucho énfasis en transiciones o eventos externos.

CAPÍTULO V

5.1 DISEÑO ESTRUCTURADO

"Diseño estructurado es el proceso de decidir que componentes, y la interconexión entre los mismos, para solucionar un problema bien especificado".

El diseño es una actividad que comienza cuando el analista de sistemas ha producido un conjunto de requerimientos funcionales lógicos para un sistema, y finaliza cuando el diseñador ha especificado los componentes del sistema y las relaciones entre los mismos.

Frecuentemente analista y diseñador son la misma persona, sin embargo es necesario que se realice un cambio de enfoque mental al pasar de una etapa a la otra. *Al abordar la etapa de diseño, la persona debe quitarse el sombrero de analista y colocarse el sombrero de diseñador.*

Una vez que se han establecido los requisitos del software (en el análisis), el diseño del software es la primera de tres actividades técnicas: *diseño, codificación, y prueba.* Cada actividad transforma la información de forma que finalmente se obtiene un software para computadora válido.

Mediante alguna metodología (en nuestro caso, estructurada basada en el flujo de información) se realiza el diseño estructural, procedimental, y de datos.

El ***diseño de datos*** transforma el modelo del campo de información, creado durante el análisis, en las estructuras de datos que se van a requerir para implementar el software.
El ***diseño estructural*** define las relaciones entre los principales elementos estructurales del programa. El objetivo principal del diseño estructural es desarrollar una estructura de programa modular y representar las relaciones de control entre los módulos.
El ***diseño procedimental*** transforma los elementos estructurales en una descripción procedimental del software. El diseño procedimental se realiza después de que se ha establecido la estructura del programa y de los datos. Define los algoritmos de procesamiento necesarios.

Concluido el diseño se genera el código fuente y para integrar y validar el software, se llevan a cabo pruebas de testeo.

Objetivos Del Diseño Estructurado
"El diseño estructurado, tiende a transformar el desarrollo de software de una práctica artesanal a una disciplina de ingeniería".

- Eficiencia
- Mantenibilidad
- Modificabilidad
- Flexibilidad
- Generalidad
- Utilidad

"Diseño" significa planear la forma y método de una solución. Es el proceso que determina las características principales del sistema final, establece los límites en performance y calidad que la mejor implementación puede alcanzar, y puede determinar a que costos se alcanzará.

El diseño se caracteriza usualmente por un gran número de decisiones técnicas individuales. En orden de transformar el desarrollo de software en una disciplina de ingeniería, se debe sistematizar tales decisiones, hacerlas más explícitas y técnicas, y menos implícitas y artesanales.

Un ingeniero no busca simplemente *una* solución, busca la *mejor* solución, dentro de las *limitaciones* reconocidas, y realizando *compromisos* requeridos en el trabajo del mundo real.

En orden de convertir el diseño de sistemas de computadoras en una disciplina de ingeniería, previo a todo, debemos definir *objetivos técnicos claros* para los programas de computadora. Es esencial además comprender las *restricciones* primarias que condicionan las soluciones posibles.

Para realizar decisiones concisas y deliberadas, debemos identificar los *puntos de decisión* .

Finalmente necesitamos una *metodología* que nos asista en la *toma de decisiones*.

Dadas estas cosas: objetivos, restricciones, decisiones reconocidas, y una metodología efectiva, podemos obtener soluciones de ingeniería, y no artesanales.

Diseño estructurado y calidad del software

Un concepto importante a clarificar es el de *calidad.* Desafortunadamente, muchos diseñadores se conforman con un sistema que "funcione" sin reparar en un *buen* sistema.

Una corriente de pensamiento estima que un programa es bueno si sus algoritmos son astutos y no obvios a otro programador; esto refleja la "inteligencia" del programador. Otra escuela de pensamiento asocia calidad con incremento de la velocidad de ejecución y disminución de los requerimientos de memoria central. Estos son aspectos de un concepto más amplio: *eficiencia*. En general, se busca diseños que hagan un uso inteligente de los *recursos*. Estos recursos no incluyen solamente procesador y memoria, también incluyen almacenamiento secundario, tiempo de periféricos de entrada salida, tiempo de líneas de teleproceso, tiempo de personal, y más.

Otra medida de calidad es la *confiabilidad.* Es importante notar que si bien la confiabilidad del software puede ser vista como un problema de depuración de errores en los programas, es también un problema de diseño. La confiabilidad se expresa en como MTBF (mean time between fairules: tiempo medio entre fallas).

Un concepto muy relacionado a la confiabilidad y de suma importancia es el de *mantenibilidad*. Podemos definir la mantenibilidad como:

Mantenibilidad del sistema = MTBF / (MTBF + MTTR)

donde:
MTBF: tiempo medio entre fallas
MTTR: tiempo medio de reparación (mean time to repair)

Diremos que un sistema es mantenible si permite la detección, análisis, rediseño, y corrección de errores fácilmente.

En tanto la mantenibilidad afecta la viabilidad del sistema en un entorno relativamente constante, la *modificabilidad* influye en los costos de mantener un sistema viable en condiciones de cambio de requerimientos. La modificabilidad es la posibilidad de realizar modificaciones y extensiones a partes del sistema, o agregar nuevas partes con facilidad (no corrección de errores).

En estudios realizados se determinó que las organizaciones abocadas al procesamiento de datos invierten aproximadamente un 50% del presupuesto en mantenimiento de los sistemas, involucrando esto corrección de errores y modificaciones, razón por la cual la mantenibilidad y la modificabilidad son dos objetivos primarios en el diseño de software.

La *flexibilidad* representa la facilidad de que el mismo sistema pueda realizar variaciones sobre una misma temática, sin necesidad de modificaciones.

La *generalidad* expresa el alcance sobre un determinado tema.

Flexibilidad y generalidad son dos objetivos importantes en el diseño de sistemas del tipo de propósitos generales.

La *utilidad* o facilidad de uso es un factor importante que influye en el éxito del sistema y sus aceptación por parte del usuario. Un sistema bien diseñado pero con interfaces muy “duras” tiende a ser resistido por los usuario.

Finalmente diremos que eficiencia, mantenibilidad, modificabilidad, flexibilidad, generalidad, y utilidad, son componentes de la *calidad* objetiva de un sistema.

En términos simples también diremos que nuestro objetivo primario es obtener sistemas de *costo mínimo.* Es decir, es nuestro interés obtener sistemas económicos para desarrollar, operar, mantener y modificar.

El concepto de Cajas Negras

Una caja negra es un sistema (o un componente) con entradas conocidas, salidas conocidas, y generalmente transformaciones conocidas, pero del cual no se conoce el contenido en su interior.

En la vida diaria existe innumerable cantidad de ejemplos de uso cotidiano: una radio, un televisor, un automóvil, son cajas negras que usamos a diario sin conocer (en general) como funciona en su interior. Solo conocemos como controlarlos (entradas) y las respuestas que podemos obtener de los artefactos (salidas).

El concepto de caja negra utiliza el principio de *abstracción.* Este concepto es de suma utilidad e importancia en la ingeniería en general, y por ende en el desarrollo de software. Lamentablemente muchas veces para poder hacer un uso efectivo de

determinado módulo, el diseñador debe revisar su contenido ante posibles contingencias como ser comportamientos no deseados ante determinados valores. Por ejemplo es posible que una rutina haya sido desarrolla para aceptar un determinado rango de valores y falla si se la utiliza con valores fuera de dicho rango, o produce resultados inesperados. Una buena documentación en tales casos, es de utilidad pero no transforma al módulo en una verdadera caja negra. Podríamos hablar en todo caso de “cajas blancas”.

Los módulos de programas de computadoras pueden variar en un amplio rango de aproximación al ideal de caja negra. En la mayoría de los casos podemos hablar de “cajas grises”.

Recordando el modelo fundamental del sistema (diagrama de flujo de datos del nivel 0) la información entra y sale en una forma del mundo exterior por ejemplo algunas formas de información del mundo exterior son las teclas pulsadas sobre un teclado de un terminal los tonos sobre una línea telefónica y los dibujos de un monitor gráficos de computadora. Tales datos externos deben ser convertidos a una forma interna adecuada para el procesamiento. La información entra al sistema como flujo entrante. En el interior del software se produce una transición. Los datos entrantes pasan a través de un centro de transformación, moviéndose a lo largo de camino flujo saliente. El flujo de datos globales ocurre de forma secuencial y sigue uno o unos pocos caminos directos. Cuando un segmento de un diagrama de flujo de datos exhibe estas características, lo que tenemos es flujo de transformación.

El modelo fundamental del sistema implica un flujo de transformación; por tanto, todo flujo de datos se puede clasificar en esa categoría. Sin embargo, a menudo, el flujo de información esta caracterizado por un único elemento de datos denominados transacción que desencadena otro flujo de datos a través de uno entre varios caminos.

El flujo de transacción se caracteriza por el movimiento de datos a través de un camino de llegada (también denominado camino de recepción) que convierte la información del mundo exterior en una transacción. Se evalúa la transacción y, de acuerdo con su valor, el flujo sigue por unos de los muchos caminos de acción. El centro de flujo de información desde él emana los caminos de acción se denomina centro de traslación.

Hay que tener en cuenta que el DFD de un gran sistema, pueden estar presentes los 2 tipos de flujo, de transformación y de transacción. Por ejemplo dentro de un flujo de transacción el flujo de la información a través de un camino de acción puede tener características de flujo de transformación.

El diseño comienza con una evaluación del diagrama de flujo de datos de nivel 2 o 3. Se establece el tipo de flujo de información(es decir, flujo de transformación o de transacción) y se definen los limites del flujo que delimitan el centro de transformación o de transacción. De acuerdo con la situación de los limites, se convierten las transformaciones (las "burbujas" del DFD) en módulos, como estructuras de programa. La organización y la definición precisas de los módulos se realizan mediante una distribución descendente del control en la estructura (llamada factorización) y la aplicación de criterios para conseguir una modularidad efectiva. Sin embargo, pueden existir variaciones y adaptaciones. Después de todo, el diseño de software exige un conocimiento humano que frecuentemente trasciende las "reglas" de un método.

Heurísticas de diseño

Una vez desarrollada la estructura del programa, usando el diseño orientado al flujo de datos, se puede conseguir una modularidad efectiva manipulando la estructura con el siguiente conjunto de heurísticas:

1) EVALUAR LA ESTRUCTURA DE PROGRAMA PRELIMINAR PARA REDUCIR EL ACOPLAMIENTO Y MEJORAR LA COHESIÓN: Con la estructura del programa desarrollada, los módulos pueden expandirse o reducirse para la mejora de la independencia de los módulos. Un módulo expandido se convierte en dos o más módulos existe un componente de procesamiento común y puede redefinirse como un módulo cohesivo aparte. Un módulo reducido es la combinación del procesamiento implicado en dos o más módulos. Cuando se espera un alto acoplamiento se pueden juntar módulos para reproducir el paso de control, las referencias a datos globales y la complejidad de las interfaces.

2) INTENTAR MINIMIZAR LAS ESTRUCTURAS CON ALTO GRADO DE SALIDA; FOMENTAR UN ALTO GRADO DE ENTRADA CONFORME AUMENTE LA PROFUNDIDAD:
Buscar una distribución razonable del control. La estructura tiene forma oval que nos índica que hay varias capas de control y un gran uso de módulos de los niveles inferiores.

3) MANTENER EL EFECTO DE UN MODULO DENTRO DEL ÁMBITO DE CONTROL DE ESE MODULO: El ámbito de efecto de un módulo M se define por todos los módulos que son afectados por una decisión hecha en el módulo M. El ámbito de control del módulo M está dado por todos los módulos que están directamente o subordinados al módulo M.

4) DEFINIR MÓDULOS CUYAS FUNCIONES SEAN PREDECIBLES, PERO EVITAR MÓDULOS QUE SEAN DEMASIADO RESTRICTIVOS: Un módulo es predecible cuando puede ser tratado como una caja negra o una abstracción procedimental que no es más que una secuencia de instrucciones que tienen una función limitada y específica. Un módulo que restringe el procesamiento a una subfunción sencilla exhibe una alta cohesión y es bien visto por un diseñador. Un módulo que restrinja arbitrariamente el tamaño de una estructura de datos requerirá de un mantenimiento para suprimir esas restricciones.

5) EVALUAR LAS INTERFASES DE LOS MÓDULOS PAR REDUCIR LA COMPLEJIDAD Y LA REDUNDANCIA Y MEJORAR LA CONSISTENCIA: La complejidad de las interfaces entre módulos es la causa de los errores del software, estas deben diseñarse solo para pasen información y deben ser consistentes con la función del módulo. La inconsistencia en las interfaces nos índica que hay una baja cohesión y por lo tanto el módulo debe ser reevaluado.

6) FOMENTAR MÓDULOS CON ENTRADA Y SALIDA ÚNICA EVITANDO LAS CONEXIONES PATOLÓGICAS: Esta heurística previene contra el acoplamiento por contenido que es cuando un módulo usa información de datos o de control que están dentro de los límites de otro módulo. El software es más fácil de comprender y mantener cuando se entra por el comienzo y se sale por el final. Las conexiones patológicas son rama o referencia en la mitad de un módulo.

7) EMPAQUETAR EL SOFTWARE DE ACUERDO CON LAS RESTRICCIONES DEL DISEÑO Y LOS REQUISITOS DE PORTABILIDAD: El empaquetamiento se refiere a las técnicas usadas para ensamblar el software en un entorno de procesamiento específico. Las restricciones de diseño hacen que un programa tenga que solaparse consigo mismo en memoria. Cuando esto vaya a ocurrir se debe reorganizar la estructura del diseño para agrupar los módulos por su grado de repetición, frecuencia de acceso y su intervalo entre llamadas.

5.2 DISEÑO ORIENTADO A OBJETOS

El diseño orientado a objetos (DOO) crea una representación del campo del problema del mundo real y la hace corresponder con el ámbito de la solución que es el Software, además produce un diseño que intercambia objeto de datos y operaciones de procesamiento en una forma que modulariza la información y el procesamiento en lugar de dejar aparte el procesamiento.

El DOO se caracteriza por mantener a la hora del diseño:

- La abstracción
- La modularidad
- El ocultamiento de la información

Todos los métodos de diseño tratan de mantener estas tres características presentes pero el DOO consigue las tres características sin complejidad.

Algunos especialista en la materia dicen acerca de la Metodología: "Ya no es necesario esforzarse en resolver un determinado problema mediante la aplicación de la estructura de datos usando las normas de los lenguajes de programación, deseando lograr mantener los principios de diseño, pues hacerlo mediante ese método nos dificultaría el trabajo, ya que en DOO muy fácil lograrlo sin tanta complejidad manteniendo las normas necesaria para el diseño".

El análisis, el diseño y la programación orientado a los objetos comprenden la INGENIERÍA DEL SOFTWARE para la construcción de un sistema orientado a los objetos.

Orígenes del diseño orientado a los objetos

En los primeros días de la informática, los lenguajes ensambladores permitían a los programadores utilizar instrucciones Máquina, para manipular elementos de datos, pero el nivel de abstracción era muy bajo.

Al parecer algunos lenguajes de programación de alto nivel (Fortran, cobol) lo lograron con la presencia de estructura de datos y de control logrando detallarlo mediante procedimiento, fue entonces cuando evolucionaron los conceptos de refinamiento de funciones y modularidad. Después en los años setenta aparecen los conceptos de abstracción y el ocultamiento de la información aunque los diseñaron se centraban

todavía en los procesos y sus representaciones, pero al mismo tiempo los lenguajes se enriquecieron con sus estructuras y tipos de datos (PASCAL).

Entonces los lenguajes convencionales surgidos de la tradición fueron evolucionando (Fortran, Algol), los investigadores trabajaban en una nueva clase de lenguaje de simulación y de creación de prototipos (SIMULA, SMALLTALK) el énfasis estaba en la abstracción de datos y se representaba por un conjunto de objeto de datos y un conjunto correspondiente de operaciones los cuales se usaban distintamente a los lenguajes convencionales.

Los primeros trabajos de la abstracción en los últimos 20 años sentaron las bases al establecer la importancia de la abstracción, del ocultamiento de la información y de la modularidad para la calidad del software.

Seguidamente en los años ochenta la rápida evolución del Fortran, Ada, seguida de un crecimiento explosivo de los dialectos de C como C++, produjeron un interés inusitado en el DOO

Conceptos de diseño orientado a los objetos

El DOO introduce un conjunto de términos, notaciones y procedimientos para la derivación del diseño del software. Para conseguir un DOO debemos establecer un mecanismo para:

- Representar la estructura de datos
- Especificar el proceso
- Llevar a cabo el procedimiento de invocación

Objetos, operaciones y mensajes

Objetos: es un componente del mundo real que se ha hecho corresponder con el campo del software. En un sistema el objeto es considerado como un productor o consumidor de información

Ejemplo: monitores, interruptores, archivos.

Operaciones: consiste de una estructura de datos privada y en procesos, cuando se hace corresponde un objeto con su realización software, que pueden legítimamente transformar la estructura de datos, se conocen como métodos y servicios.

Mensaje: Las operaciones contiene unas instrucciones procedimentales y de control que pueden ser invocadas mediante un mensaje, en otras palabras, es una petición al objeto para que realice alguna de sus operaciones

El objeto tiene una parte compartida que es su interfaz, los mensajes llegan al objeto a través de su interfaz y especifica la operación que se desea que realizar sobre el objeto, aunque no como se ha de realizar.

Al definir un objeto como parte privada y proporcionar mensajes para invocar el procesamiento adecuado, conseguimos el ocultamiento de la información, es decir, dejar oculto para el resto de los elementos del programa los detalles de implementación del objeto.

Aspectos de diseño

Existen cinco criterios para juzgar la capacidad del método de diseño de conseguir la modularidad y los relaciona con el DOO:

Descomponibilidad : facilidad del método de diseño de conseguir un gran problema en sub-problemas más fáciles de resolver

Compatibilidad : grado en que se asegura que las componentes del programa una vez diseñada pueda ser rehusadas para crear programas.

Comprensibilidad : facilidad con que se comprende una componente de un programa sin necesidad de buscar información en otros módulos.

Continuidad : capacidad de realizar cambios en un programa y que se manifiesten por sí mismo con solo unos cambios correspondientes en unos pocos módulos

Protección : reduce la propagación de efectos colaterales cuando se produce un error en un modulo.

A partir de estos criterios se sugiere la derivación de cinco principios básicos de diseño para arquitectura modulares:

- Unidades lingüísticamente modulares: que corresponden a unidades sintácticas en el lenguaje utilizado, es decir el lenguaje de programación que se vaya a utilizar debe soportar la modularidad.
- Pocas interfaces: se debe minimizar el número de interfaces entre módulos.
- Interfaces pequeñas: minimizar la cantidad de información que se mueve a través de una interfaz.
- Interfaces explícitas: deben de comunicarse de forma obvia y directa cada vez que los módulos que se quiera modificar.
- Ocultamiento de la infamación: Esto ocurre cuando toda la información de un modulo esta oculta para un acceso desde el exterior.

Los criterios y los principios se aplican a cualquier método de diseño pero como veremos más adelante el DOO los aplica más eficientemente.

Clases instancias y herencia

Todos los objetos son miembros de una clase mayor y heredan la estructura de datos privadas y las operaciones que han sido definidas para esa clase. Dicho de otra forma, una clase es un conjunto de objetos que tiene las mismas características. Así, un objeto es una instancia de clase mayor.

El uso de clases, subclases y herencia es de crucial importancia en la ingeniería del software moderna. La reutilización de componentes de software de software (lo que nos permite conseguir la composición) se lleva a cabo creando objetos (instancias) que se forman sobre atributos y las operaciones existentes heredadas de una clase o de una subclase.

A diferencia de otros conceptos de diseño que son independientes del lenguaje de programación que se utilice, la implementación de las clases, de las subclases y de los objetos varía de acuerdo con el lenguaje de programación que se utilice. Por esta razón, el anterior tratamiento genérico requerirá ciertas modificaciones para el contexto de cada lenguaje de programación específico. Por ejemplo, en Ada, los objetos se implementan como paquetes y las instancias se obtienen mediante el uso de abstracciones de datos y tipos.

Descripciones de los objetos

La descripción de diseño de un objeto puede tener dos formas distintas.

Una descripción del protocolo: que establece la interfaz del objeto, definiendo cada mensaje que puede recibir el objeto y las operaciones que realiza el objeto cuando recibe el mensaje.

Una descripción de la implementación: que demuestra los detalles de cada operación implicada en la recepción de un mensaje por el objeto. Los detalles de implementación incluyen la información sobre la parte privada del objeto, esto es, los detalles internos de la estructura de datos, y los detalles procedimentales que describen las operaciones.

Para grandes sistemas con muchos mensajes, a menudo se pueden crear categorías de mensajes. Por ejemplo, para el sistema hogar seguro las categorías de mensaje podría ser:

- Mensajes de configuración del sistema.
- Mensajes de monitorización.
- Mensajes de sucesos.

Una descripción de la implementación está compuesta por la siguiente información:

- una especificación del nombre del objeto y de la referencia a una clase.
- una especificación de la estructura de datos privada, con indicación de los elementos de datos y sus tipos.
- una descripción procedimental de cada operación o, alternativamente, referencia a esas descripciones procedimentales.

Las diferencias entre la información contenida en la descripción del protocolo y la contenida en la descripción de la implementación, en términos de los "usuarios" y de los "suministradores" de los servicios, es la siguiente.

Un usuario de un "servicio" proporcionado por un objeto debe familiarizarse con el protocolo de invocación de ese servicio, es decir, con la forma de especificar qué es lo que desea.

Al suministrador del servicio (el propio objeto), lo que le concierne es cómo proporcionar el servicio al usuario, es decir, los detalles de implementación.

Métodos de diseño orientados a los objetos

Esencialmente, el análisis orientado a los objetos (AOO) es una actividad de clasificación. Es decir, se analiza un problema con el fin de determinar clases de objetos que sean aplicables en el desarrollo de la solución. El diseño orientado a los objetos (DOO) permite al ingeniero de software indicar los objetos que se derivan de cada clase y las inter-relaciones entre ellos. Además el DOO debe proporcionar una notación que refleje las relaciones entre los objetos.

A inicios de los años ochenta, tanto Abbot como Booch establecen que el DOO debe comenzar con una descripción en lenguaje natural de la estrategia de solución, mediante una realización en software, de un problema del mundo real.

En su estado actual de evolución, los métodos de DOO combinan elementos de las tres categorías de diseño presentadas anteriormente: diseño de datos, diseño arquitectónico y diseño procedimental. Al identificar clases y objetos, se crean abstracciones de datos. Asociando operaciones a los datos, se especifican módulos y se establece una estructura para el software. Al desarrollar mecanismos para la utilización de los objetos se describen las interfaces.

Los métodos de DOO están caracterizados por los siguientes pasos:

1. Definir el problema.
2. Desarrollar una estrategia informal (narrativa de procesamiento) para la realización en software del campo del problema del mundo real.
3. Formalizar la estrategia mediante los siguientes sub-pasos:
    - Identificar los objetos y sus atributos.
    - Identificar las operaciones que se les puede aplicar a los objetos.
    - Establecer interfaces que muestren las relaciones entre los objetos y las operaciones.
    - Decidir aspectos del diseño detallado que proporcionen una descripción de la implementación de los objetos.
4. Volver a aplicar los pasos 2, 3 y 4 recursivamente.
5. Refinar el trabajo realizado en el AOO, buscando las subclases, las características de los mensajes y otros detalles de elaboración.
6. Representar la(s) estructura(s) de datos asociado(s) con los atributos del objeto.
7. Representar los detalles procedimentales asociados con cada operación.

Definición de clases y objetos

La aplicación de los principios y métodos de análisis de requisitos permite al analista y al diseñador llevar a cabo dos sub-pasos necesarios:

- Describir el problema.
- Analizar y clarificar las limitaciones conocidas.

Refinamiento de las operaciones

Una vez que se ha identificado los objetos del espacio de la solución, el diseñador selecciona el conjunto de operaciones que actúan sobre los objetos.

Las operaciones se identifican examinando todos los verbos de la estrategia informal (La narrativa de la estrategia informal).

Aunque existen muchos tipos distintos de operaciones generalmente se pueden dividir en tres grandes categorías:

- Operaciones que manipula los datos de alguna forma (Por eje: adiciones, eliminaciones, cambio de formato, selecciones etc.).
- Operaciones que realizan algún cálculo.
- Operaciones que monitorizan un objeto frente a la ocurrencia de algún suceso de control.

Por ejemplo la narrativa de procesamiento de Hogar Seguro contiene los siguientes fragmento de frase:"Cada sensor tiene asignado un número y un tipo" y "se programa una contraseña maestra para activar y desactivar el sistema".

Estas dos frases indican tres cosas:

- Que para el objeto sensor es relevante una operación de asignación.
- Que se aplicará una operación programar al objeto sistema.
- Que activar y desactivar son operaciones que se aplican al sistema; también que el estado del sistema tiene que ser definido(usando la notación diccionario de datos) como:

ESTADO DEL SISTEMA =[ACTIVADO | DESACTIVADO]

La operación programar queda asignada durante el AOO, pero es durante el DOO cuando se refina en varias operaciones más especificas. Que son requeridas durante para configurar el sistema. Por ejemplo, tras dialogar con el ingeniero del producto, con el analista y probablemente con el departamento de ventas, el diseñador puede reelaborar la narrativa de procesamiento original y escribir lo siguiente para programar:

Programar permite al usuario de Hogar Seguro configurar el sistema una vez que ha sido instalado. El usuario puede:

- Instalar números de teléfonos.
- Definir retardos para alarmas.
- Construir una tabla de sensores que contenga el ID, el tipo y la ubicación de cada sensor.
- Cargar una contraseña maestra.

Por tanto el diseñador ha definido la operación simple programar y la ha sustituido por las operaciones: instalar, definir, construir y cargar. Cada una de las nuevas operaciones entra a formar parte de objeto sistema. Conoce las estructuras internas de datos internas que implementan los atributos del objeto y se invocan mandando al objeto mensajes de la forma:

MENSAJE(sistema) -> instalar: RECIBE número de teléfono;

Que implica que, para proporcionar al sistema un número de teléfono de emergencia, se le debe enviar al objeto sistema el mensaje instalar.

Componentes de programas e interfaces

Un aspecto muy importante de la calidad del diseño de software la modularidad, esto es, la especificación de componentes de programas (módulo) que, combinados forman el programa completo. El enfoque orientados a los objetos define el objeto como componente de programa que se encuentra enlazado con otros componentes. (Por ejemplo: datos privados, operaciones). Pero la definición de objeto y operaciones no es suficiente. Durante el diseño también debemos identificar las interfaces que existen entre los objetos y la estructura general (en sentido arquitectónico) de objetos.

Aunque un componente de programa es una abstracción de diseño, se puede representar dentro del contexto del lenguaje de programación que se va ha utilizar para implementar el diseño. Para dar cabida al DOO, el lenguaje de programación que se use en la implementación debe permitir crear el siguiente componente de programa (modelizado a partir de ADA)

```
PACKGE nombre-de-componente-de-programa IS
    TYPE especificación de los objetos de datos
            .
            .
            .
    PROC especificación de las operaciones asociadas...
PRIVATE
    PROC operación.1 (descripción de la interfaz) IS
            .
            .
            .
    END
    PROC operación.n (descripción de la interfaz) IS
            .
            .
            .
     END
END nombre-de-componente-de-programa
```

De acuerdo con el anterior LDP (lenguaje de diseño de programa) similar a ADA, un componente de programa queda especificado indicando tanto los objetos de datos como las operaciones. La parte de especificación de componente indica todos los objetos de datos (declaración de la sentencia TYPE) y todas las operaciones(PROC por “procedimiento”) que actúan sobre ellos. La parte privada (PRÍVATE) del

componente proporciona los detalles ocultos de la estructura de datos y del procesamiento. En el contexto de la discusión anterior, el paquete (PACKAGE) es conceptualmente similar a los objetos tratados en este CAPÍTULO.

El primer componente de programa a identificar debe ser el módulo de más alto nivel desde el que se origina todo el procesamiento y del que evolucionan todas las estructuras de datos.

Una notación para el DOO

Se han propuesto otras notaciones que a menudo se encuentran en el campo industrial. En esta sección repasaremos algunas de ellas. Booch propone una notación que combina cuatro diagramas distintos para la creación del DOO:

- Un diagrama de clases: Que refleja las clases y sus relaciones.
- Un diagrama de objetos: Representa objetos específicos(instancias de una clase) y los mensajes que pasan por ellos. Como parte del diseño físico, se asignan las clases y los objetos a componentes de software específicos.
- El diagrama de módulos: a veces denominado diagrama de BOOCH sirve para ilustrar esos componentes de programas.
- Diagrama de procesos: Que permite al diseñador reflejar como se asignan los procesos a procesadores concretos en un gran sistema, Debido a que muchos sistemas orientados a los objetos contienen varios programas que se pueden ejecutar en varios procesadores distribuidos.

Representación de relaciones entre clases y entre objetos

BOOCH sugiere diagramas que representan clases y objetos, así como las relaciones entre ellos.

El diagrama de clases indica la “arquitectura de clases de un sistema”

La herencia se representa con un símbolo de flecha.

Las conexiones de doble trazo indican que una clase utiliza información contenida en otra clase.

Los símbolos de los extremos de la notación (doble trazo) indican la cardinalidad. Además del propio diagrama de clases, se puede definir cada clase con una plantilla que incluya su nombre, sus atributos y sus operaciones. Así como la información sobre la herencia. Los parámetros y el comportamiento.

La clase se define como parte del diseño de un sistema y existen independientemente del comportamiento en ejecución del sistema. Sin embargo los objetos se crean y se eliminan dinámicamente a medida que se ejecuta el sistema.

Diagrama de objeto: Refleja cada instancia de una clase, las operaciones que se le aplican y los mensajes que se le pasan.

Las líneas que conectan a los objetos y terminan con un recuadro etiquetado indican el tipo de datos que contiene el mensaje.

La flecha paralela a la línea indica el tipo de sincronización entre los objetos (Asíncrono, Asíncrono, Retardado).

Para cada objeto se crea una plantilla de objeto que define el objeto y los mensajes que se pasan.

Modularización del diseño

Ya hemos hablado de la importancia de la modularización en el diseño de software. Pero aquí surge una pregunta

¿Cómo se define el concepto de modulo cuando se aplica el DOO?

La respuesta a esta pregunta nos lleva a una notación de diseño que asigna clases y objetos a los componentes físicos de programan (módulos) y por tanto representa una visión de implementación de la arquitectura del programa.

El diagrama de módulos representa un componente de programa (objeto) como una caja dividida en una parte de especificación (visible al exterior) y una parte privada(parte del cuerpo) en esta etapa todavía no se especifican los detalles de implementación de la parte privada o cuerpo del paquete los objetos de datos se representan con un ovalo alargado mientras que las operaciones que actúan sobre ellos se representan mediante rectángulos, las flechas de conexión implican independencia, es decir el paquete o componente en el origen de la flecha depende del paquete o componente de programa en la punta de la flecha. El componente de programa de mayor nivel A, depende de los objetos y las operaciones contenidas en B, C, D necesarios para satisfacer su función.

El Software de un sistema basado en computadora es ejecutado por uno o más procesadores (computadora) que interactúan con otros dispositivos mediante una serie de conexiones definidas.

El uso del diseño y de la implementación orientada a los objetos no altera esa situación. Por ello se usa el diagrama de proceso, para indicar los procesadores (Caja sombreada) los dispositivos (Cajas en blanco) y las conexiones (líneas) que definen la arquitectura del hardware de un sistema.

En el diagrama también se indican los procesos software que se ejecutan en cada procesador.

Una estrategia alternativa de diseño orientado a los objetos

El método de diseño orientado a los objetos esta orientado al desarrollo del software con lenguaje de programación tipo Ada.

El método no se centra explícitamente sobre varios conceptos importantes de la orientación a los objetos esta sección describe un enfoque alternativo para el DOO que ha evolucionado del desarrollo del software con lenguajes de programación tipo Smalltalk a lenguajes que soportan directamente la abstracción, la herencia, los mensajes y todos los demás conceptos del DOO.

El enfoque de DOO es apropiado para el diseño preliminar el objetivo principal es definir y caracterizar las abstracciones de una forma que resulte en una definición de todos los objetos. LORENSE sugiere el siguiente enfoque:

1- Identificar las abstracciones de datos para cada sub-sistema las abstracciones de datos son las clases del sistema a partir del documento de requisitos, se debe llevar acabo el proceso de abstracción de forma descendente. A menudo las clases corresponden a objetos físicos del sistema que se modeliza.

2- Identificar los atributos de cada abstracción los atributos son variables de instancia de cada clase muchas veces, cuando las clases corresponden a objetos físicos las variables de instancia requeridas serán obvias.

3- Identificar las operaciones de cada abstracción. las operaciones son los métodos o procedimientos de cada clase. en este momento no se especifican los detalles de implementación de los métodos, sino solo su funcionabilidad. Si una nueva abstracción hereda de otra clase, hay que inspeccionar los métodos de esa otra clase para ver si alguno es redefinido en la nueva clase.

4- Identificar la comunicación entre objetos este paso define los mensajes que se envían unos objetos a otros. Aquí se define la correspondencia entre los métodos y los mensajes que invocan a los métodos.

5- Probar el diseño con guiones los guiones, consistentes en conjuntos de mensajes a objetos prueban la habilidad del diseñador de satisfacer la especificación de requisitos del sistema.

6- Aplicar herencia donde sea apropiado si el proceso de abstracción de datos del paso 1. se ha realizado de forma descendente, se puede especificar la herencia allí, sin embargo si las abstracciones se han creado de abajo hacia arriba también se aplica la herencia.

Integración del DOO con el análisis estructurado y el diseño estructurado

Actualmente, los investigadores nos se han puesto de acuerdo al respecto de sí es posible la integración de estas dos estrategias de análisis y diseño en un solo método único. Algunos piensan que las dos estrategias son muy diferentes, mientras que otros creen que se pueden utilizar elementos de ambos métodos para desarrollar un modelo completo del problema. Primero, se tratara el área donde ambos métodos tienen alguna similitud y convergencia conceptual, o sea situaciones en las que pueden coexistir pacíficamente ambas técnicas.

Los elementos claves de AE/DE son:

- El modelo de flujo.
- Diccionario de datos.
- La especificación del control.
- La especificación de procesos.
- Un diagrama de entidad-relación (E- R), para la modelización de datos.

Aunque el diagrama de flujos de datos pareciera que podría proporcionar una buena conexión entre el AE y el DOO no es así, ya que sus elementos de datos flujos (flechas) y almacenes (líneas dobles) y los procesos ("burbujas") constituyen un modelo aproximado de los objetos y sus operaciones, respectivamente. Pero no es así.

Recordemos que la orientación a los objetos crea una representación del campo del problema del mundo real y la hace corresponder con el ámbito de la solución que es el software. Por esta razón, las representaciones de AE/DE que reflejan los elementos del mundo real relativos al problema proporcionarían un mejor puente con el AOO/DOO. Las entidades externas (cuadros) que representan a los productores y consumidores del flujo de datos será posible candidato a objetos. Los objetos de datos definidos como parte del diagrama E-R también son candidatos a objetos. Es cierto que los DFDs de menor nivel podrían proporcionar una indicación de los atributos de algunos objetos y que los procesos indicados a esos diagramas podrían ser operaciones aplicadas a los objetos. Pero un DFD no refleja un sistema desde el punto de vista orientado a los objetos.

La especificación de control (CSPEC) proporciona un modelo de comportamiento que puede ayudar al diseñador a comprender mejor el paso de mensajes en el sistema orientado a los objetos. Finalmente, el diagrama de entidad relación puede ayudar a aislar clases y subclases candidatas, así como las relaciones entre ellas.

Ward describe una correspondencia entre el AE/DE para sistema de tiempo real y el diseño orientado a los objetos. Se puede representar una abstracción de datos (una clase) como una transformación incorpórea con sus entradas y sus salidas; no es instanciada incorporándola al modelo de flujo; tal transformación podría construirse teniendo en cuenta la reusabilidad, incluyendo datos y transformaciones adecuadas para muchos ámbitos de aplicación (en los que aparezca la abstracción de datos). Esencialmente, lo que Ward sugiere es que se puede representar una clase o una subclase como una burbuja aparte. Los flujos hacia y desde esa burbuja siguen representando flujo de datos, pero implican operaciones asociadas a la clase. Para instanciar un objeto, se incluye la burbuja independiente en el contexto del modelo de flujo. Por ejemplo, una clase sensor puede estar modelizada como una abstracción de datos (utilizando un diagrama E-R) y representada por una burbuja aparte (incorpórea). Se podría instanciar un objeto de la clase sensor (por ejemplo, sensor de humo) en un modelo de flujo especifico, incluyendo una burbuja sensor de humo en el DFD.

Algunos investigadores piensan que no hay una oposición clara entre el AE/DE y el DOO, mientras que otros (los puristas) creen que los métodos orientados a los objetos y el enfoque de diseño estructurado se incluyen mutuamente (y que el DOO es con mucho superior). Solo el tiempo dirá si los dos valiosos métodos de diseño siguen caminos diferentes o si llegan a un matrimonio duradero.

Resumen

El diseño orientado a los objetos crea un modelo del mundo real que puede ser realizado en software. Los objetos proporcionan un mecanismo para representar el ámbito de información, mientras que las operaciones describen el procesamiento asociado con el ámbito de información. Los mensajes (un mecanismo de interfaz) proporcionan el medio por el que se invocan las operaciones. La característica distintiva de AOO/DOO es que los objetos saben que operaciones se pueden aplicar sobre ellos. Este conocimiento se consigue combinando abstracciones de datos y de procedimientos en un solo componente de programa (denominado objeto o paquete).

AOO/DOO ha evolucionado como resultado de una nueva clase de lenguajes de programación orientados as los objetos, tales como Smalltalk, C++, Objetive-C, CLU, Ada, Modula y otros. Consecuentemente, las representaciones del diseño orientado a los objetos son mas adecuados que otras que dependen del lenguaje de programación.

La Metodología AOO/DOO consiste en tres pasos que requieren que el diseñador establezca el problema, defina una estrategia informal de resolución y formalice la estrategia, identificando los objetos y operaciones, especificando interfaces y proporcionando detalles de implementación para la abstracción de datos y de procedimientos. El papel del DOO es tomar las clases y los objetos básicos definidos en el AOO y refinarlos con los detalles adicionales de diseño.

Los diseños se representan mediante alguna de las notaciones gráficas existentes y el lenguaje de diseño de programas.

El diseño orientado a los objetos representa un enfoque único para los ingenieros de software.

Diagrama de Clases

Muestra el conjunto de clases y objetos importantes que hacen parte de un sistema, junto con as relaciones existentes entre estas clases y objetos. Muestra de una manera estática la estructura de información del sistema y la visibilidad que tiene cada una de las clases, dada por sus relaciones con las demás en el modelo.

Especificación de Operación

El conjunto de operaciones describen el comportamiento de los objetos de una clase. La sintaxis de una operación en UML es:

visibility name (parameter-list) : return-type-expression { property-string }
Cada uno de los parámetros en *parameter-list* se denota igual que un atributo. Los demás elementos son los mismos encontrados en la notación de un atributo

Diagrama de Componentes

Un diagrama de componentes muestra las dependencias lógicas entre componentes software, sean éstos componentes fuentes, binarios o ejecutables. Los componentes software tienen tipo, que indica si son útiles en tiempo de compilación, enlace o ejecución. Se consideran en este tipo de diagramas solo tipos de componentes. Instancias específicas se encuentran en el diagrama de ejecución.

Se representa como un grafo de componentes software unidos por medio de relaciones de dependencia (generalmente de compilación). Puede mostrar también contenencia de entre componentes software e interfaces soportadas.

Diagrama de Despliegue

Un diagrama de despliegue muestra la configuración de los elementos de procesamiento en tiempo de ejecución y los componentes software, procesos y objetos que se ejecutan en ellos. Instancias de los componentes software representan manifestaciones en tiempo de ejecución del código. Componentes que solo sean utilizados en tiempo de compilación deben mostrarse en el diagrama de componentes.

Un diagrama de despliegue es un grafo de nodos conectados por asociaciones de comunicación. Un nodo puede contener instancias de componentes software, objetos, procesos (un caso particular de un objeto). Las instancias de componentes software pueden estar unidos por relaciones de dependencia, posiblemente a interfaces.

CAPÍTULO VI

6.1 REVISIÓN DE LOS CONCEPTOS DE BASE DE DATOS

Una base de datos es una colección de elementos interrelacionados que pueden procesarse por uno o más sistemas de aplicación. Un sistema de base de datos está formado por una base de datos, por un sistema computacional de propósito general - llamado sistema de gestión de bases de datos (SGBD) - que manipula la base de datos, así como por el hardware y el personal apropiados.

Un sistema de base de datos, adecuadamente diseñado, integra los datos comunes a varias unidades funcionales de la compañía y facilita su manipulación. Además de simplificar la inserción, la eliminación y la modificación cotidianas de los registros, los sistemas de base de datos facilitan la identificación y la cuantificación de las relaciones derivadas entre los elementos de los datos, la recopilación de la información en resúmenes estadísticos, la inferencia sobre las posibles tendencias del negocio y otras operaciones.

Mediante tales facilidades, el sistema de base de datos transforma los datos puros en información.

Componente de un Sistema de Base de Datos

El Hardware

El hardware es el conjunto de dispositivos físicos sobre los que reside una base de datos. Consiste en una o más computadoras, unidades de disco, vídeo-terminales, impresoras, unidades de cinta magnética, cables de conexión y otros equipos auxiliares y de conexión del equipamiento.

El Software

Un sistema de base de datos incluye dos tipos de software:

- El software de propósito general para la gestión de base de datos, comúnmente llamado sistema de gestión de base de datos (SGBD o DBMS).

- El software de aplicación, que usa las facilidades del SGBD para manipular la base de datos con el fin de llevar a cabo una función específica de la compañía, tal como la emisión de los estados o el análisis de las tendencias de las ventas.

El software de aplicación generalmente se escribe por los empleados de una compañía para resolver un problema específico. Usa las facilidades del SGBD para el acceso a y la manipulación de los datos en la base de datos, proporcionando los informes o los documentos necesarios para los requisitos de información y de procesamiento de la compañía.

El sistema de gestión de base de datos (SGBD) es un software, parecido a un sistema operativo o a un compilador, que brinda un conjunto de servicios a los usuarios finales, los programadores y otros.

Un SGBD típicamente brinda la mayoría de los siguientes servicios:

- Herramientas para la definición y el control centralizado de los datos, conocida como diccionario de datos/directorio (DD/D) o catálogo.
- Mecanismos de seguridad e integridad de datos.
- Acceso concurrente a los datos para varios usuarios.
- Utilidades para la consulta, la manipulación y la elaboración de informes orientados al usuario.
- Utilidades para el desarrollo de sistemas de aplicación orientados al programador.

Los Datos

Obviamente, ningún sistema de base de datos puede existir sin los datos, los hechos básicos sobre los que se fundamentan las necesidades de información y de procesamiento de una compañía.

Sin embargo, el factor esencial a considerar es que los datos que conforman una base de datos tiene que ser cuidadosa y lógicamente estructurados.

Las funciones del negocio deben analizarse, los elementos de los datos y las interrelaciones deben identificarse y definirse cabalmente y estas definiciones deben almacenarse de manera precisa en el diccionario de datos.

Las Personas

Se puede identificar dos tipos de personas:

- Usuarios: Los ejecutivos, los gerentes, los administradores, el personal de oficina.
- Profesionales de la computación: Los administradores de la base de datos, los analistas, los programadores, los diseñadores del sistema y de la base de datos, los administradores de los sistemas de información.

En la actualidad las organizaciones dependen muchos de su base de datos e incluso hasta define el éxito en las decisiones que se tengan que tomar.

El término compartir datos sugiere que personas en áreas funcionales diferentes compartan un grupo común de datos, cada cual para sus propias aplicaciones.

El efecto de combinar los datos en una base de datos produce sinergia; es decir, los datos combinados tienen más valor que la suma de ellos en los archivos por separados.

Esto no sólo permite que cada grupo continúe teniendo acceso a sus datos, sino que, bajo límites razonables de control, también pueda tener acceso a los otros datos.

Conceptos Básicos

Dato: Son hechos representan la materia prima de la información

Puede ser : Elemental

Agrupado

Campo: Son los lugares donde guardamos los datos

Registro: Es la información que se tiene acerca de una entidad u objeto. Es un conjunto de datos

Puede ser: * Lógico * Físico

* Longitud fija * Longitud variable

Archivo: Es el conjunto de un mismo tipo de registros (lógico) dado.

Campo Clave: Es el atributo o conjunto de atributos que utiliza la computadora para identificar un registro o tuple.

Tipos de Archivos por su Función

Maestro: Almacenan la información más importante de la empresa

Transacciones: Son archivos temporales que almacenan información que se registra en un determinado periodo

Reporte: Contienen los resultados obtenidos, después de ejecutar un programa

Tabla: Almacenan tablas de referencias

Respaldo: Son copias de protección de algunos archivos

6.2 NORMALIZACIÓN

Codd propone un proceso llamado de normalización de las relaciones, el cual es un método propio del modelo relacional y que consiste en descomponer las relaciones originales en otras más pequeñas con el fin de eliminar una serie de anomalías de almacenamiento y manipulación que se pueden dar en las relaciones iniciales y que conformarían la futura base de datos relacional.

Antes de introducir el proceso de normalización, se estudiará el siguiente ejemplo.

Ejemplo. Sea un sistema hospitalario, lo referente a la atención de pacientes así como los médicos asignados. Esta situación se representa por medio de la relación siguiente:

ATENCIÓN-DE-PACIENTES

Cédula	Nom_Paciente	Dirección	Núm_Médico	Nom_Médico	Esp_Médico	Ubicación_Médico
23376	Carlos	Rivas	125	Mario	Neurología	H.México
32158	María	Carazo	125	Mario	Neurología	H.México
46859	Pedro	León	130	Carmen	Cardiología	H.Militar
17856	Adrián	M.Limón	135	Isabel	Hematología	HEODRA
24631	Juana	Estelí	135	Isabel	Hematología	HEODRA
67913	Cecilia	Rama	135	Isabel	Hematología	HEODRA

Esta estructura tiene una serie de problemas relacionados con inconsistencias y anomalías de administración de las tuplas. Esto es, al insertar, modificar o suprimir tuplas dentro de esta relación, se van a presentar problemas.

En efecto, supóngase que se debe ingresar una nueva paciente de la Mina el Limón llamada Astrid y que debe ser atendida en neurología. Entonces se le asigna el médico Mario. Esta situación obliga a insertar la tupla

[81325, Astrid, M.Limón, 125, Mario, Neurología, H.México].

De esta forma, se debe repetir la especialidad del médico así como el hospital donde labora.

Por otra parte, si el paciente Pedro es dado de alta y se borra la tupla correspondiente, se estaría perdiendo la información de que Carmen es una cardióloga y que trabaja en el Hospital Militar.

Finalmente, si Isabel es reemplazada por Erika, se deben hacer varias modificaciones en diferentes tuplas.

Para evitar este tipo de anomalías, se utiliza el proceso de normalización, que consiste en obtener una serie de relaciones, a partir de la relación original sin perder el contenido inicial, es decir, que si se aplica el operador join a las relaciones resultantes, se obtiene la relación original.

Así una mejor solución al problema de la relación ATENCION-DE-PACIENTES sería contar con dos relaciones, una con los datos referentes a los pacientes y otra con los datos referentes a los médicos, con un atributo común, que en este caso sería Núm_Médico.

PACIENTE

Cédula	Nom_Paciente	Dirección	Núm_Médico
23376	Carlos	Rivas	125
32158	María	Carazo	125
46859	Pedro	León	130
17856	Adrián	M.Limón	135
24631	Juana	Estelí	135
67913	Cecilia	Rama	135

MEDICO

Núm_Médico	Nom_Médico	Esp_Médico	Ubicación_Médico
125	Mario	Neurología	H.México
130	Carmen	Cardiología	H.Militar
135	Isabel	Hematología	HEODRA

En este caso, si se desea introducir la paciente Astrid, solo debe hacerse una inserción en la relación PACIENTE, mediante la tupla [81325, Astrid, M.Limón, 125].

Por su parte, si el paciente Pedro es dado de alta, se borra la tupla en la relación PACIENTE y no se pierde información de Carmen en la relación MEDICO.

Finalmente, si Isabel es reemplazada, los cambios solo se hacen a nivel del Núm_Médico.

También es importante notar que, bajo esta nueva circunstancia, se puede tener un conjunto de médicos, no todos asignados en un momento dato. En el caso anterior esto no era posible.□

El proceso de normalización se compone de una serie de seis etapas llamadas *formas normales*. El objetivo de este proceso es llevar a que las relaciones presentes en la base de datos se encuentren en la quinta forma normal.

Beneficios que se obtienen de una base de datos correctamente normalizada:

- Reducir los problemas asociados con supresión e inserción de tuplas.
- Reducir el tiempo asociado con modificaciones de tuplas.
- Identificar problemas potenciales que pueden requerir un análisis adicional.
- Mejorar la información para la toma de decisiones referentes a la organización física de los datos.

Primera Forma Normal (1FN)

La primera forma normal se refiere a la representación de una relación, en la cual los atributos son diferentes y los valores de cada uno de esos atributos son componentes atómicos.

Ejemplo. Sea la relación
EMPLEADO(Número, FechaPago1, Monto1, FechaPago2, Monto2, FechaPago3, Monto3)

EMPLEADO

Número	Fp1	M1	Fp2	M2	Fp3	M3
123	01/05/98	5000	01/06/98	5500	01/07/98	6000
124	01/05/98	7000	01/06/98	7200	01/07/98	7400
125	01/05/98	6000	01/06/98	7000	01/07/98	8000

A pesar de que todos los atributos tienen nombres diferentes, en realidad las fechas de pago representan una sola fecha; lo mismo con los atributos montos. El problema de una relación como esta es el hecho que, por ejemplo, para cada nuevo mes se deben crear dos nuevos atributos. En este caso, la relación solo funciona para un trimestre.

Así, para transformar una relación en una que se encuentre en 1FN, se podría considerar la siguiente relación:

EMPLEADO(Número, Fecha_Pago, Monto)

EMPLEADO

Número	Fecha_Pago	Monto
123	01/05/98	5000
134	01/05/98	7000
245	01/05/98	6000
123	01/06/98	5500
134	01/06/98	7200
245	01/06/98	7000
123	01/07/98	6000
134	01/07/98	7400
245	01/07/98	8000

Dependencias funcionales

Así como se producen relaciones o asociaciones entre relaciones, es posible establecer relaciones o asociaciones entre varios atributos de una misma relación, llamadas dependencias. Así, los valores de algunos atributos en una relación pueden determinar en forma única, el conocimiento de los valores de otros atributos de la misma relación.

Estas dependencias traducen reglas semánticas que son muy comunes en el modelaje del mundo real.

Definiciones básicas

Dependencia Funcional. Formalmente, sea R un esquema de relación y sean X, Y subconjuntos de atributos de R. Se dice que existe una dependencia funcional entre X y Y, y se denota por $X \rightarrow Y$, si para cualquier tuplas t1 y t2 de una relación R de este esquema tal que si t1[X] = t2[X] entonces t1[Y] = t2[Y].

En este caso, a X se le llama determinante y a Y el dependiente.

Ejemplo. Sea el esquema de relación

CHOFER(Cédula, Nombre, Dirección, Fecha_Ingreso, #Placa)

CHOFER

Cédula	Nombre	Dirección	Fecha_Ingreso	#Placa
45689	Juan Mora	Chinandega	12/06/78	7-985-6
14527	Juan Mora	Matagalpa	10/12/80	1-854-3
54683	María Salinas	León	10/12/80	1-235-6
45734	Carlos Mata	Somoto	12/11/86	6-479-1

En este caso se pueden establecer varios hechos:

Para cada número de cédula, existe un único nombre asociado, es decir, se verifica la dependencia funcional Cédula $\rightarrow$ Nombre.

Puesto que un camión puede ser conducido por varios chóferes no se verifica la dependencia funcional entre #Placa y Cédula. En este caso se representa

#Placa /$\rightarrow$ Cédula.

Se tiene que Nombre /$\rightarrow$ #Placa, puesto que dos personas distintas pueden tener el mismo nombre. Sin embargo, {Nombre, Fecha_Ingreso} $\rightarrow$ #Placa, se verifica, si se está seguro que dos personas con el mismo nombre no fueron contratadas el mismo día.

Es importante mencionar que la dependencia funcional es un concepto que deriva del significado de los datos y no del comportamiento de una relación dada.

Axiomas de inferencia de Amstrong

Sean X, Y, Z subconjuntos de atributos de una relación R, en donde se verifican las dependencias funcionales $X \rightarrow Y$ y $Y \rightarrow Z$. Entonces, las siguientes reglas se cumplen:

A1	Reflexividad	$X \rightarrow X$
A2	Aumento	$X \rightarrow Y \Rightarrow X \cup Z \rightarrow Y$
A3	Transitividad	$\{ X \rightarrow Y, Y \rightarrow Z \} \Rightarrow X \rightarrow Z$
A4	Unión	$\{ X \rightarrow Y, X \rightarrow Z \} \Rightarrow X \rightarrow Y \cup Z$
A5	Descomposición	$X \rightarrow Y \Rightarrow X \rightarrow Z$ con $Z \subseteq Y$
A6	Pseudotransitividad	$\{ X \rightarrow Y$ y $Y \cup Z \rightarrow W \} \Rightarrow X \cup Z \rightarrow W$

Derivación de una dependencia funcional

Sea F un conjunto de dependencias funcionales, en donde U es el conjunto de atributos involucrados en F. Se dice que la dependencia funcional g: $X \rightarrow Y$ se deriva de F y se denota por $F \vdash g$, si existen n dependencias funcionales $f_1, f_2, ..., f_n$ tal que

1. $f_n = g$
2. $\forall i, i \in \{1, ..., n\}, f_i \in F$ o bien f_i se infiere de $\{f_1, f_2, ..., f_{i-1}\}$

Usando las reglas A1, A2 y A3.

Conjunto de dependencias funcionales o Cerradura

El conjunto de todas las dependencias funcionales de una relación, llamado cerradura, denotado por F^+, se define como

$$\mathbf{F^+ = \{ f / F \vdash f \}}$$

Por esta definición podemos ver que el conjunto de dependencias funcionales es siempre un subconjunto de su cerradura, es decir, $\mathbf{(F^+)^+ = F^+}$.

Saturación

Sea F un conjunto de dependencias funcionales. Se llama la saturación de un conjunto X con respecto a F, y se denota por X^+, al conjunto de atributos que son determinados por las dependencias funcionales que se derivan de F, es decir,

$$\mathbf{X^+ = \{ A / F \vdash X \rightarrow A \}}$$

Utilizando la saturación de un conjunto de atributos, se puede mostrar el siguiente resultado

$X \rightarrow Y \in F^+ \Leftrightarrow Y \subseteq X^+$,el cual es muy interesante y expresa que para determinar si una dependencia funcional se puede derivar de un conjunto F de dependencias funcionales,

es equivalente a mostrar que el dependiente de la dependencia funcional es subconjunto de la saturación del determinante.

Dependencias funcionales equivalentes

Sean F y G dos conjuntos de dependencias funcionales. Se dice que F y G son equivalentes, y se denota por $F \equiv G$, si sus coberturas son iguales, es decir,

$$\mathbf{F \equiv F \text{ si } F^+ = G^+}$$

En este caso se dice que G es una **cobertura** de F.

Un conjunto de dependencias funcionales F se dice **no redundante**, si no existe un subconjunto propio G de F, tal que $F^+ = G^+$

Sea F un conjunto de dependencias funcionales. Sea F* el conjunto de las dependencias de la cerradura de F, en donde el determinado contiene un solo atributo, es decir, son **dependencias elementales**. Entonces (F*)+ = F+

Con esta definición se puede definir el concepto de cobertura mínima.

Una **cobertura** G de F, se dice que es **mínima** si

1. Las dependencias funcionales de G son elementales
2. No existe subconjunto propio H de G tal que $H^+ = F^+$

Teorema de descomposición

Sea un esquema de relación R(X, Y, Z) con X, Y y Z conjuntos de atributos de R, tal que la dependencia funcional $X \rightarrow Y$ se verifica en R. Entonces, la relación R se descompone en las relaciones R1 = R[X, Y] y R2 = R[X, Z], es decir,

$$\mathbf{R = R1*R2}$$

Ejemplo. Considerar el esquema de relación

CURSO(Grupo, Código, Profesor, Aula, Día, Hora)

CURSOS

Grupo	Código	Profesor	Aula	Día	Hora
01	C-1122	González	H-06	Lunes	09
03	C-2123	Helo	M-01	Lunes	14
04	C-3344	Helo	H-06	Martes	10
07	C-1515	Araya	H-06	Jueves	16

Según este esquema se puede decir que para un profesor y un código dados, solo existe un aula, un día y una hora asociadas. En este caso, se puede decir que se establece la dependencia funcional

Código, Profesor → Aula, Día, Hora.

Si se aplica el teorema de descomposición usando esta dependencia funcional, se obtienen las siguientes relaciones:

R1 = R[Código, Profesor, Aula, Día]
R2 = R[Grupo, Código, Profesor]

Es interesante notar que la descomposición engendra redundancia. En efecto, si la dependencia funcional X → Y se verifica, el conjunto de atributos que conforman X, se duplica en las dos relaciones que surgen de la descomposición. Es el único tipo de redundancia que será tolerada en las diferentes etapas de normalización y se le denomina **emigración de atributos**.

Utilizando el concepto de dependencia funcional, se puede definir una **llave** de una relación de la siguiente forma: Sea R(U) un esquema relacional y sea X un subconjunto de atributos de U. Se dice que X es una llave de R(U), si

1. La dependencia funcional X → U se verifica en R y,
2. Ningún subconjunto propio de X puede determinar funcionalmente a U.

Segunda Forma Normal

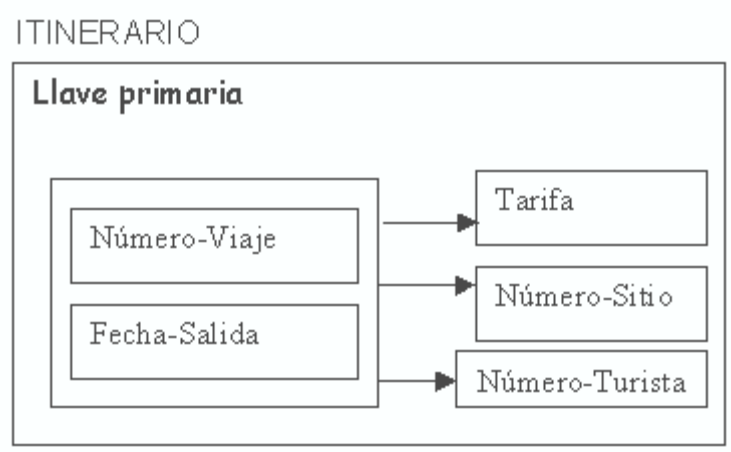
Sea un esquema de relación R, X una llave para R y A un atributo no llave. Se dice que el atributo **A depende parcialmente de X** si se verifica Y → A, en donde Y es un subconjunto propio de la llave X.

Una relación se dice que se encuentra en su segunda forma normal (2FN), si se encuentra en 1FN y si ningún atributo no llave depende parcialmente de la llave primaria.

Ejemplo. Considerar la siguiente relación

ITINERARIO(Número-Viaje, Fecha-Salida, Tarifa, Número-Sitio, Número-Turista)

Puesto que cada viaje tiene asociado una sola tarifa, en esta relación se verifica la siguiente dependencia funcional: Número-Viaje → Tarifa, según se muestra en la figura a continuación



Este es un ejemplo de una relación que no está en 2FN, ya que el atributo Tarifa depende parcialmente de la llave primaria que es la combinación de los atributos Número-Viaje, Fecha-Salida.

Debido a este hecho se presentan una serie de anomalías de manipulación de datos. En efecto, si se desea introducir un valor de Número-Viaje, Tarifa se puede hacer solo si existe un valor correspondiente para la llave primaria Número-Viaje, Fecha-Salida.

Por otra parte, si se suprime un valor de la llave se puede perder un valor único de Número-Viaje, Tarifa.

Finalmente, si se desea modificar un valor de la Tarifa asociado a un valor de un Número-Viaje se puede tener inconsistencias o un alto costo de actualización para el caso de los valores duplicados.

Con el fin de eliminar estos problemas se aplica el teorema de descomposición a la dependencia funcional que viola la 2FN, en este caso Número-Viaje $\rightarrow$ Tarifa. Al aplicar dicho teorema, se obtienen las siguientes relaciones:

ITINERARIO1(Número –Viaje, Fecha-Salida, Número-Sitio, Número-Turista) y COSTO-ITINERARIO(Número-Viaje, Tarifa).

Tercera Forma Normal

Antes de introducir el concepto de tercera forma normal, es preciso definir previamente, el término de dependencia transitiva.

Sea R (X,Y,Z) un esquema de relación, en donde X, Y y Z son subconjuntos de atributos. Se dice que Z es trasitivamente dependiente de X si existe Y tal que se dan las siguientes condiciones:

- Se verifica $X \rightarrow Y$,
- No se verifica $Y \rightarrow X$,
- Se verifica $Y \rightarrow Z$.

Se dice que una relación R se encuentra en tercera forma normal (3FN) si se encentra en 2FN y no existe una dependencia transitiva entre atributos no llave.

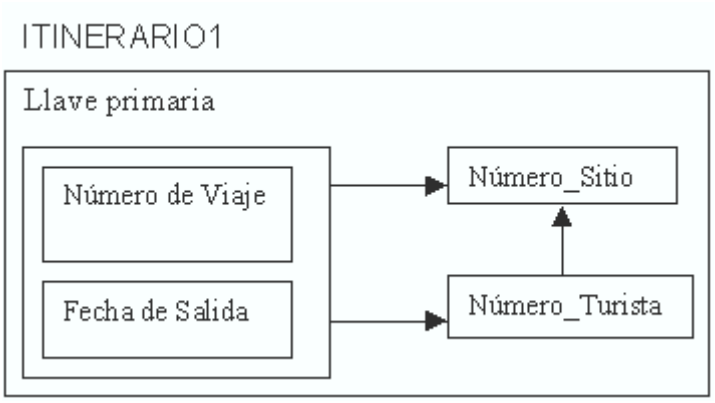
Ejemplo. Considerar la relación siguiente:

ITINERARIO1 (Número-Viaje, Fecha-Salida, Número-Sitio, Número-Turista)

Se va a suponer que cada turista visita un solo sitio durante el año. Esto significa que la siguiente dependencia funcional se verifica en la relación INTINERARIO1:

Número-Turista $\rightarrow$ Número-Sitio.

Así el atributo Número-Sitio depende transitivamente de la llave y por lo tanto no se encuentra en 3FN, según se aprecia en la siguiente figura.



Al no encontrarse la relación ITINERARIO1 en tercera forma normal, se presentan varios problemas.

En efecto, si por ejemplo, se desea introducir un valor de Número-Turista, Número-Sitio, solo se puede hacer si existe un valor asociado de Número-Viaje, Fecha-Salida.

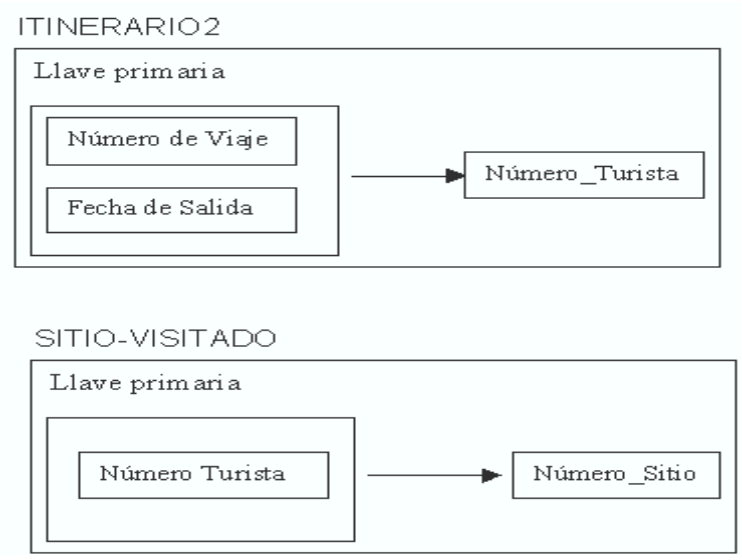
Por su parte, si se suprime un valor de Número-Turista, Número-Sitio.

Asimismo, si se desea modificar un valor Número-Turista, Número-Sitio, se puede tener un alto costo de actuación debido a la redundancia presente.

Para resolver el problema, se debe aplicar el teorema de descomposición a la dependencia funcional transitiva para obtener las dos relaciones.

ITINERARIO2 (Número-Viaje,Fecha-Salida, Número-Turista)
SITIO-VISITADO(Número-Turista, Número-Sitio),

Según se aprecian en la figura siguiente.



TERCERA FORMA NORMAL BOYCE-CODD (3FNBC)

Una relación se encuentra en tercera forma normal Boyce-Codd (3FNBC) si todos los atributos son determinados solo por llaves, es decir, si cada vez que

$$X \rightarrow A, A \notin X,$$

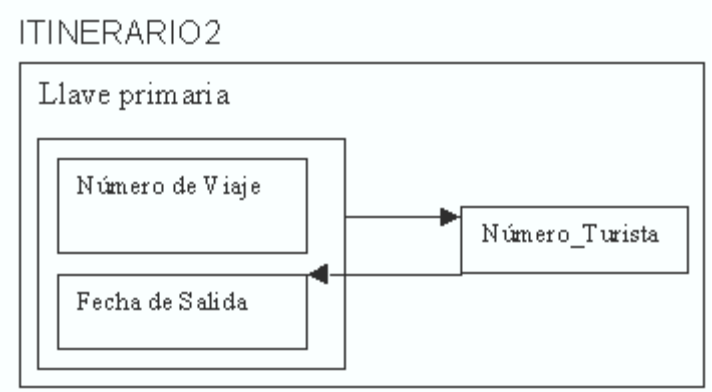
se verifica en R, entonces X contiene una llave de R.

Ejemplo. Considerar la relación ITINERARIO2 y suponer que se tiene la siguiente regla de integridad: ¨Cada turista tiene una sola fecha de salida¨.

Esta regla de integridad se puede traducir por medio de la siguiente dependencia funcional:

Número-Turista→ Fecha-Salida.

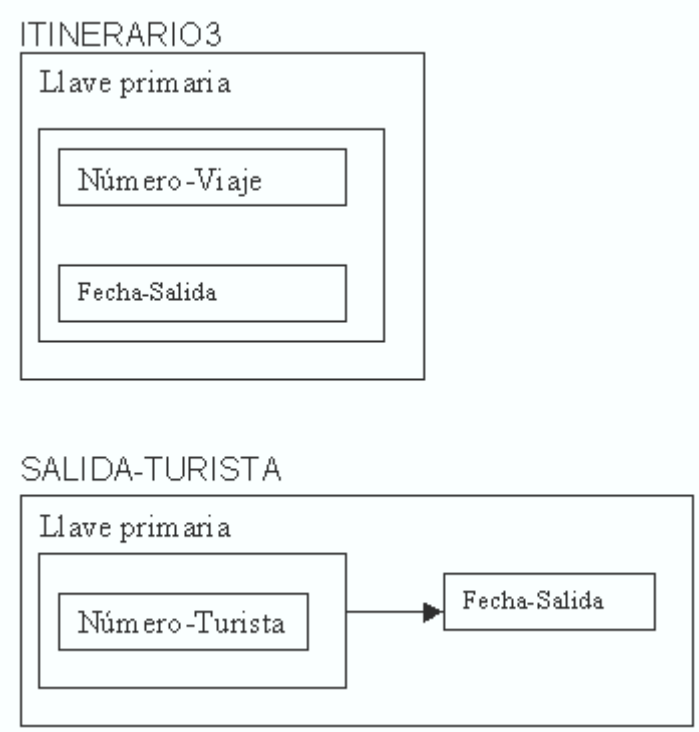
Según se aprecia en la figura



Así, ITINERARIO2 no se encuentra en 3FNBC. En esta situación aun persisten anomalías de actualización. En efecto, si por ejemplo, se desea suprimir un número de viaje en una fecha dada, se puede perder la información sobre cuándo el turista hará el viaje. Entonces, para resolver estos problemas, se aplica el teorema de descomposición a la dependencia funcional Número-Turista→Fecha-Salida, para obtener las dos relaciones siguientes:

INTINERARIO3(Número-Viaje, Número-Turista)
SALIDA-TURISTA(Número-Turista, Fecha-Salida),

Según se aprecia en la figura siguiente.



DEPENDENCIAS MULTIVALUADAS Y LA CUARTA FORMA NORMAL (4FN)

Muchas personas dedicadas al análisis y diseño de bases de datos relacionales afirman que es suficiente contar con una base de datos cuyas relaciones se encuentren en 3FNBC pues se contaría con un alto grado de consistencia semántica. Sin embargo, existen ejemplos de relaciones que, a pesar de encontrarse en 3FNBC, pueden provocar problemas de almacenamiento, como se verá en el siguiente ejemplo.

Ejemplo. Considerar la relación siguiente:

DISPONIBILIDAD(Nombre-Turista, Continente, Ciudad)

Sobre la disponibilidad de los turistas del Club de Ecoturismo en visitar sitios en determinados continentes y las ciudades de las cuales pueden salir.

En la figura de abajo, se aprecia un ejemplo de una tal relación.

DISPONIBILIDAD

Nombre_Turista	Continente	Ciudad
Carlos	América	Lima
Carlos	América	Caracas
Carlos	América	México
Pierre	América	México
Pierre	América	Bogotá
Carlos	África	Lima
Carlos	Arica	Caracas
Carlos	Arica	México
Pierre	Asia	México
Pierre	Asia	Bogotá

Se puede ver que esta relación se encuentra en 3FNBC, pues la llave primaria la conforman todos los atributos. Sin embargo, se pueden notar varias anomalías de almacenamiento.

En efecto, por ejemplo, no es posible insertar un continente y un turista sin conocer la ciudad de la cual puede salir, pues todos los atributos conforman la llave y como se sabe, no es permitido tener valores nulos en una llave primaria. Lo mismo ocurre con un turista y una ciudad, sin conocer el continente que se puede visitar.

Suponga que se va a insertar la tupla [Pierre, América, Santiago] . Pero, pues que desde Santiago, entonces la inserción de la tupla anterior, debe desencadenar la inserción de la tupla [Pierre, Asia, Santiago].

Además, qué ocurre si se deben suprimir los continentes visitados por un turista en particular? En este caso, se suprime también el conocimiento que se tiene entre los

turistas y las ciudades. Finalmente, si una ciudad de un turista cambia, se debe hacer la modificación varias tuplas.

El problema con esta relación es que no se encuentra en la cuarta forma normal.

Sin embargo, previo a la introducción del concepto de cuarta forma normal, se requiere la definición de lo que se entiende por dependencias multivaluadas. Estas se pueden ver como una generalización de las dependencias funcionales.

En efecto, se puede decir que una dependencia multivaluada existe entre dos conjuntos de atributos X y Y, y se denota por

$$\mathbf{X \rightarrow\!\!\!\rightarrow Y}$$

si solo el conocimiento de X, e independientemente de otros atributos, determina un conjunto de valores relativos a Y.

Formalmente, sean X, Y subconjuntos de atributos de la relación R.

Se dice que la *dependencia multivaluada* $X \rightarrow\!\!\!\rightarrow Y$ se verifica si cada vez que las tuplas

$$\mathbf{[x_1, y_1, z_1] \text{ y } [x_2, y_2, z_2]}$$

se encuentran en R, entonces deben estar también las tuplas

$$\mathbf{[x_1, y_2, z_1 \ ] \text{ y } [x_1, y_1, z_2]}$$

En forma análoga a las dependencias funcionales, se puede introducir sobre las dependencias multivaluadas, una serie de axiomas de inferencia. Estas reglas fueron introducidas por C. Berri, R. Fagin y J.H.

Howard en [BERR77] y se presentan a continuación.

Axiomas de inferencia de las dependencias multivaluadas

Sean X,Y,V,W subconjuntos de atributos de una relación R, y Z el completo de los atributos de X U Y con respecto a R. Entonces se verifican las siguientes reglas:

1. Reflexividad	$X \rightarrow\!\!\!\rightarrow X$
2. Complemento	$X \rightarrow\!\!\!\rightarrow Y \Rightarrow X \rightarrow\!\!\!\rightarrow Z$
3. Aumento	$X \rightarrow\!\!\!\rightarrow Y$ y $V \subseteq W \Rightarrow X \cup W \rightarrow\!\!\!\rightarrow Y \cup V$
4. Unión	$\{ X \rightarrow\!\!\!\rightarrow Y, X \rightarrow\!\!\!\rightarrow Z \} \Rightarrow X \rightarrow\!\!\!\rightarrow Y \cup Z$
5. Transitividad	$\{ X \rightarrow\!\!\!\rightarrow Y, Y \rightarrow\!\!\!\rightarrow Z \} \Rightarrow X \rightarrow\!\!\!\rightarrow Z - Y$
6. Pseudotransitividad	$\{ X \rightarrow\!\!\!\rightarrow Y, Y \cup W \rightarrow\!\!\!\rightarrow Z \} \Rightarrow X \cup W \rightarrow\!\!\!\rightarrow Z - (Y \cup W)$
7. Descomposición	$\{ X \rightarrow\!\!\!\rightarrow Y, X \rightarrow\!\!\!\rightarrow Z \} \Rightarrow \{ X \rightarrow\!\!\!\rightarrow Y \cap Z, \quad X \rightarrow\!\!\!\rightarrow Y - Z, X \rightarrow\!\!\!\rightarrow Z - Y \}$

Por otra parte, al igual que las dependencias funcionales, se puede enunciar un teorema de descomposición para el caso de las dependencias multivaluadas:

Teorema de Descomposición: Sea una relación R(X,Y,Z), en donde X, Y y Z son conjuntos de atributos y tal que la dependencia multivaluada $X \rightarrow\rightarrow Y$ se verifica en R. Entonces R se puede descomponer en las relaciones $R_1 = R[X, Y]$ y $R_2 = R[X,Z]$, es decir, $R = R_1 * R_2$.

Por su parte, se dice que una relación R(X,Y,Z) se encuentra en *cuarta forma normal* (4FN) si se encuentra en 3FNBC y si cada vez que una dependencia multivaluada

$X \rightarrow\rightarrow Y$ se verifica, entonces X contiene a una llave de R.

Con respecto al ejemplo anterior, se puede ver que la dependencia multivaluada Nom-Turista $\rightarrow\rightarrow$ Ciudad-Salida se verifica en la relación DISPONIBILIDAD y por lo tanto no se encuentra en 4FN. Si se aplica el teorema de descomposición a esta dependencia multivaluada, se obtienen las dos relaciones que aparecen en la figura siguiente.

DISP-TURISTA

Nombre-Turista	Continente
Carlos	América
Pierre	América
Carlos	Africa
Pierre	Asia

CIUDAD-TURISTA

Nombre-Turista	Ciudad
Carlos	Lima
Carlos	Caracas
Carlos	México
Pierre	México
Pierre	Bogotá

QUINTA FORMA NORMAL (5FN)

La quinta forma normal (5FN) se refiere a las llamadas dependencias producto que garantizan la descomposición de una relación en tres o más relaciones, manteniendo el contenido original y con menor redundancia.

Formalmente, sean X_1, X_2, ..., X_n, subconjuntos de atributos de una relación R, en donde la unión es igual a los atributos de R. Se dice que *la dependencia producto de orden n,* denotada por

$$*[X_1] [X_2] \ldots [X_n]$$

se verifica en R si

$$R = R[X_1] * R[X_2] * \ldots * R[X_n]$$

Una relación R se dice que se encuentra en *quinta forma normal* (5FN) si cada dependencia producto $*[X_1]\ [X_2]\ \ldots\ [X_n]$ de R está inducida por las llaves candidatas de R, es decir, cada X contiene una llave candidata de R.

Ejemplo. Considerar la siguiente relación:

GUIA (Nombre-Guía, Nombre-Sitio, Ciudad-Salida,)

Que traduce los desplazamientos de los Guías del Club de Ecoturismo y que se presenta en la figura siguiente.

GUÍA

Nombre-Guía	Nombre-Sitio	Ciudad-Salida
Juan	Irazú	San José
Juan	Tikal	Miami
Ana	Irazú	Miami
Juan	Irazú	Miami

En este caso, se puede ver que la relación se encuentra en 4FN. Así , persisten una serie de anomalías. En efecto, si por ejemplo se desea introducir la tupla [Ana, Irazú, Miami]. Pero, si se introduce la tupla [Juan, Irazú, Miami] esto no condiciona nada.

Con respecto a las supresiones, el problema es contrario. Así, si se borra la tupla [Juan, Irazú, Miami], entonces se debe suprimir [Ana, Irazú . Miami]. Por el contrario, si se suprime la tupla [Juan, Irazú, Miami] esto no condiciona nada. El problema es que se verifica la dependencia producto:

***[Nom-Guía,Nom-Sitio],[Nom-Sitio,Ciudad-Sal], [Nom-Guía,Ciudad-Sal].**

Y esta dependencia producto no está inducida por las llaves de la relación GUIA.

Esto es, la relación GUIA no se puede descomponer en solo dos relaciones, ya que si así se hiciera se generarían tuplas ajenas a la relación original, como lo muestra la figura siguiente.

GUIA[N-G, N-S] ∗ GUIA[N-S, C-S]

Nom-Guía	Nom-Sitio	Ciudad-Salida
Juan	Irazú	San José
Juan	Irazú	Miami
Juan	Tikal	Miami
Ana	Irazú	San José
Ana	Irazú	Miami

GUIA[N-S, C-S] ∗ GUIA[N-G, C-S]

Nom-Guía	Nom-Sitio	Ciudad-Salida
Juan	Irazú	San José
Juan	Irazú	Miami
Juan	Tikal	Miami
Ana	Tikal	San José
Ana	Irazú	Miami

Por lo tanto, en este caso es recomendable descomponer la relación GUIA en las tres relaciones, según la descomposición producto presente, como lo muestra la figura siguiente.

GUIA[N-G, N-S]

Nom-Guía	Nom-Sitio
Juan	Irazú
Juan	Tikal
Ana	Irazú

GUIA[N-S, C-S]

Nom-Sitio	Ciudad-Salida
Irazú	San José
Tikal	Miami
Irazú	San José

GUIA[N-G, C-S]

Nom-Guía	Ciudad-Salida
Juan	San José
Juan	Miami
Ana	Miami

6.3 DISEÑO DE INTERFACES DE USUARIO (GUI´S)

En este tema se considera el diseño de la interfaz de usuario, un tema que se ha cada vez más importante a medida que ha aumentado el uso de las computadoras.

La interfaz en muchos sentidos es el "envoltorio" del software de computadora.

La interfaz debe tener las siguientes características:

- fácil de aprender
- simple de utilizar
- directo
- no muy estricto

En el diseño de la interfaz de usuario se debe tomar en cuenta el estudio de las o las personas que van a manejar el software así como los aspectos de la tecnología. Una de las muchas preguntas que debe plantearse el diseñador son:

1. ¿Quién es el usuario?
2. ¿Cómo aprende el usuario a interaccionar con un sistema nuevo basado en computadora?
3. ¿Qué espera el usuario del sistema?

Estas deberán ser respondidas como parte del diseño de la interfaz de usuario.

Factores humanos

- La fase "factores humanos" en el sistema interactivo basado en software tiene una gran variedad de significados:
- Debemos comprender la percepción, la psicología cognitiva de la lectura, la memoria humana y el razonamiento deductivo e inductivo.
- Debemos comprender al usuario y su comportamiento.
- Debemos comprender las tareas que el sistema basado en software realiza para el usuario como parte de la interacción entre el hombre y la máquina.

Fundamentos de la percepción humana

Un ser humano percibe el mundo a través de un sistema sensorial.

Cuando se considera la interfaz Hombre - Máquina (IHM) predominan los sistemas visual, táctil y auditivo. Esto permite al usuario de un sistema basado en computadoras percibir información, almacenarla en la memoria (Humana), y procesarla utilizando un razonamiento inductivo o deductivo.

La información que se extrae de la interfaz debe ser almacenada en la memoria humana para ser usada posteriormente. La memoria humana es un sistema extremadamente complejo la cual esta compuesta por:

- Memoria a corto plazo (MCP)
- Memoria a largo plazo (MLP)

Una IHM debe especificarse de tal manera que permita al humano desarrollar heurísticas (consejos, reglas y estrategias) para la interacción.

Nivel de habilidad humana y comportamiento

Es importante tener en cuenta las diferencias individuales de habilidad, personalidad y comportamiento entre usuarios de un sistema basado en computadora.

El nivel de habilidad del usuario final tendrá un impacto significativo sobre su habilidad para extraer información significativa de la IHM para responder efectivamente a tareas y poder aplicar de forma efectiva heurísticas que crean un ritmo de interacción.

Tareas y factores humanos

Independientemente de las diferentes tareas que realiza cada aplicación, es posible una categorización global, para ello casi siempre se realizan las siguientes tareas:

1. Tareas de comunicación
2. Tareas de diálogo
3. Tareas Cognitivas
4. Tareas de Control

Estilos de interacción entre hombres y máquinas

Los estilos de interacción Hombre - Máquina abarcan una gran variedad de opciones que están íntimamente ligadas a la evolución de las computadoras y a las tendencias de la IHM. Las opciones para el estilo de interacción han aumentado a medida que el hardware se ha ido haciendo más sofisticado

En los primeros días de las computadoras (antes de las pantallas gráficas, el ratón, las estaciones de trabajos de altas prestaciones y similares) la única forma de interacción Hombre-Máquina era la interfaz de preguntas y ordenes. La comunicación era únicamente textual y conducida mediante órdenes y respuestas a preguntas generadas por el sistema.

Una variante (y a menudo una mejora) de la interfaz de órdenes y preguntas es la interfaz de menú simple. Aquí, se presenta al usuario una lista de opciones y la decisión apropiada se selecciona mediante la introducción de un código.

Ejemplo: Elija la opción deseada:
1 = introducir datos manualmente
2 = introducir datos desde un archivo existente
3 = realizar análisis detallados
4 = generar salidas gráficas
5 = otras opciones
seleccionar opción ?_

A medida que el hardware se ha hecho más eficiente y los ingenieros de software han aprendido más sobre los factores humanos y su impacto en el diseño de la interfaz, han ido apareciendo las modernas interfaces orientadas a ventanas con opciones de señalar y elegir. Esta interfaz de tercera generación (interfaz de ventanas, iconos, menús y dispositivos de señalización) ofrece al usuario un gran número de ventajas:

Se pueden visualizar cualquier tipo de información simultáneamente, permitiendo al usuario cambiar de contexto sin perder conexión visual con los otros trabajos. Las ventanas permiten al usuario realizar muchas tareas cognitivas y de comunicación.

El esquema de menús desplegables permite realizar muchas tareas interactivas diferentes. Estos menús permiten al usuario realizar tareas de control y diálogo en forma sencilla.

La utilización de iconos gráficos, menús desplegables, botones y técnicas de presentación continua reducen el número de pulsaciones en el teclado. Esto puede suponer un aumento en la eficiencia de aquellos con poca experiencia en mecanografía.

Diseño de la interfaz hombre - máquina

El diseño de la interfaz Hombre - Máquina es un elemento de un tema más amplio que hemos denominado diseño de software.

El proceso global de diseño de una interfaz de usuario comienza con la de diferentes modelos de funcionamiento del sistema (tal y como se percibe exteriormente). Se especifican las tareas humanas y las orientadas a las computadoras, necesarias para el funcionamiento del sistema:

- Se consideran los aspectos de diseño que han de aplicarse a todos los diseños de interfaces.
- Se utilizan herramientas para realizar prototipos.
- Se implementa el modelo de diseño evaluándose el resultado bajo criterios de calidad.

Modelos de diseño de interfaces

Cuando se va a diseñar un IHM deben tenerse en cuenta cuatro modelos diferentes. El ingeniero de software crea un modelo de diseño; un ingeniero de interacción (o el ingeniero de software) establece un modelo de usuario; el usuario final desarrolla una imagen mental que denomina modelo del usuario o percepción del sistema, y los que desarrollan el sistema crean la imagen del sistema.

Un modelo de diseño del sistema completo incorpora datos, representaciones de los datos, procedimientos y estructuras del software.

El modelo de usuario representa el perfil de los usuarios finales del sistema. Además, los usuarios pueden ser categorizados como:

1. Novatos
2. Usuarios intermitentes, con capacidad de aprender
3. Usuarios frecuentes, con capacidad de aprender

La percepción del sistema (el modelo del usuario) es la imagen mental del sistema que se forma el usuario final.

La imagen del sistema combina la manifestación externa de sistema basado en computadora (el aspecto y el "sentir" de la interfaz), con toda la información de soporte (libros, manuales, cintas de vídeo) que describen la sintaxis y la semántica del sistema.

Análisis y modelización de tareas

El análisis de tareas se realiza también utilizando enfoques elaborativos u orientados a objetos que son análogos pero que se aplican a actividades humanas.

El análisis de tareas se puede aplicar en dos formas:

- Un sistema interactivo, basado en computadora se utiliza normalmente reemplazando una actividad manual o semimanual.
- El ingeniero de interacción puede estudiar una especificación existente (sistema que se cuenta actualmente) para una solución basada en computadora y desarrollar un conjunto de tareas que se adaptarán al:

1. modelo de usuario
2. modelo de diseño
3. percepción del sistema

El ingeniero de interacción debe definir y clasificar las tareas sin tener en cuenta el análisis global de las mismas.

El modelo de diseño de la interfaz debe tener en cuenta cada una de estas tareas, de tal forma que sea consistente con el modelo de usuario y la percepción del sistema.

Un enfoque alternativo para el análisis de tareas es el orientado a los objetos.

El modelo de diseño de la interfaz no propondrá una codificación literal de cada una de estas acciones, sino que definirá tareas de usuario para realizar el resultado final.

Una vez que cada tarea o acción se ha definido, comienza el diseño de la interfaz.

Los primeros pasos en el proceso del diseño se realizan siguiendo el siguiente enfoque:

- Estableces los objetivos e intenciones de cada tarea.
- Asignar a cada objetivo/intención una secuencia de acciones específicas.
- Especificar la secuencia de acciones tal y como se ejecutarán en el nivel de interfaz.
- Indicar el estado del sistema, es decir ¿Qué aspecto tiene la interfaz en el momento en que se ejecuta una acción de la secuencia.

- Definir los mecanismos de control, es decir, los dispositivos y acciones accesibles al usuario para modificar el estado del sistema.
- Indicar cómo afectan los mecanismos de control al estado del sistema.
- Indicar como interpreta el usuario el estado del sistema a partir de la información suministrada a través de la interfaz.

Aspecto de diseño

Según evoluciona el diseño de una interfaz de usuario, se muestran cuatro aspectos de diseños comunes:

- Tiempo de respuesta del sistema.
- Facilidades de ayuda al usuario.
- Manejo de la información de error.
- Asignación de nombres a las órdenes.

Herramientas de implementación

El proceso de diseño de la interfaz de usuario es interactivo.

Para facilitar este enfoque interactivo de diseño se ha desarrollado una amplia gama de herramientas de diseño y prototipo. Estas se denominan "herramientas para la interfaz de usuario o sistemas de desarrollo de interfaz de usuario" (SDIU); que proporcionan modelos u objetos que facilitan la creación de ventanas, menús, interacción con dispositivos, mensajes de error, ordenes y otros elementos de entorno interactivo.

Un SDIU proporciona mecanismos para:

- Control de los dispositivos de entrada (ratones y teclados).
- Validación de la entrada de datos del usuario.
- Manejo de errores y visualización de manejos de error.
- Realimentación.
- Ayudas e indicaciones.
- Manejo de ventanas y campos, controlando el movimiento del texto dentro de las ventanas.
- Establecimiento de conexiones entre el software de aplicación y la interfaz.
- Aislamiento de la aplicación de las funciones de manejo de la interfaz.

Evaluación del diseño

Una vez que se ha creado una versión operativa de la interfaz, está debe ser evaluada para determinar si satisface las necesidades del usuario.

Directrices para el diseño de interfaces

Las categorías de directrices para el diseño de IHM son:

1. Interacción general
2. Visualización de información
3. Entrada de datos.

Interacción general

Las directrices que se centran en la interacción general son:

- Ser consistente.
- Ofrecer una realimentación significativa.
- Preguntar por la verificación de cualquier acción destructiva no trivial.
- Permitir una vuelta atrás fácil en la ejecución de la mayoría de las acciones.
- Reducir la cantidad de información que debe ser memorizada entre acciones.
- Buscar la eficiencia en el diálogo, el movimiento y el pensamiento.
- Perdonar los errores.
- Categorizar las actividades sobre la base de su función y organizar la geografía de la pantalla convenientemente.
- Proporcionar facilidades de ayuda sensibles al contexto.
- Utilizar verbos de acción simples o frases verbales cortas para nombrar las órdenes.

Visualización de la información

La información puede ser presentada de formas diferentes:

1. Con texto, dibujos y sonidos.
2. Por posición, movimiento y tamaño
3. Utilizando colores y resolución; e incluso por omisión.

Las directrices que se centran en la visualización de información son:

- Mostrar sólo la información que sea relevante en el contexto actual.
- No abrumar al usuario con datos, sino utilizar un formato de presentación que permita una asimilación rápida de la información.
- Utilizar etiquetas consistentes, abreviaciones estándar y colores predecibles.
- Permitir al usuario mantener el contexto visual.
- Producir mensajes de error significativos.
- Utilizar mayúsculas y minúsculas, tabulaciones y agrupaciones de textos para ayudar a la comprensión.
- Utilizar ventanas (si están disponibles) para modularizar los diferentes tipos de información.
- Utilizar representaciones “analógicas” para mostrar la información que es más fácil de asimilar bajo este tipo de representación.
- Considerar la geografía disponible en la pantalla y utilizarla eficientemente.

Entrada de datos

Las siguientes directrices se centran en la entrada de datos:

- Minimizar el número de acciones de entrada de datos que debe realizar el usuario.
- Mantener la consistencia entre la información visualizada y los datos de entrada.
- Permitir al usuario personalizar la entrada de datos.
- La interacción también debe ser flexible y estar ajustada al modelo de entrada preferido por el usuario.
- Desactivar órdenes que sean inapropiados en el contexto actual.
- Permite al usuario controlar el flujo interactivo.
- Proporcionar ayuda en todas las acciones de entrada de datos.
- Eliminar las entradas innecesarias.

6.4 USABILIDAD DEL SOFTWARE

La llamada *usabilidad* del software es un tema que está adquiriendo una gran importancia en ciertos dominios, como las aplicaciones web, donde evitar ciertos errores puede ser crítico. Es igualmente trascendental en el desarrollo de los SBCs.

Se trata, en esencia, de medir la satisfacción del usuario. Para ello se suele realizar test heurísticos, a priori, que algún experto a determinado sobre el uso de la aplicación en el dominio. Pero los realmente determinantes serán las tareas de prueba que tendrán que hacer un grupo de usuarios sobre funciona miento del sistema. Éstas deben tener unos objetivos, unos márgenes de tiempo para su realización, unas medidas de satisfacción con unos criterios prefijados, etc.

Los usabilidad debería medir, fundamentalmente, cuestiones relacionadas con:

- Interfaz
- Adaptación del sistema a la tarea
- Coste del aprendizaje en el uso del sistema
- Adaptación a los estándares
- Documentación (ayudas y explicación del funcionamiento)
- Aspectos subjetivos dependientes del dominio

En cualquier caso, convendría establecer una jerarquía de los criterios correspondientes a la medida de la usabilidad. Igualmente, habría que elaborar cuestionarios con preguntas claras y concisas. Existen diversos tipos de cuestionarios y normas para su elaboración que deberían considerarse.

Utilidad

Es una medida de evaluación del plan estratégico que se estableció al principio del proyecto. Consiste en determinar si la operatividad que ofrece el sistema es la adecuada para la organización. Dada la naturaleza de esta medida es la propia organización la responsable de su evaluación.

Las cuestiones básicas que debería están relacionadas con la mejora producida en eficiencia (mejoras en productividad y tiempos de respuesta), eficacia (se reducen los casos sin resolver o mal resueltos) y la capacidad (se abordan nuevos tipos de problemas).

También en este caso habría que establecer una jerarquía que permitiera organizar los criterios que miden la utilidad. Para su evaluación se comparan dichos criterios con el nuevo software y sin él.

Un método adecuado para evaluar la utilidad es el AHP (Analytic Hierarch*y Process*) de Saaty. Permite cuantificar juicios subjetivos y fuerza al evaluador a comparar alternativas con respecto a los criterios establecidos. Tiene una serie de pasos.

CAPÍTULO VII

7.1 LA CALIDAD DEL SOFTWARE

¿Qué es Software de Calidad?

Philip Crosby, en su libro de Calidad expone:

Salvando las diferencias, la Calidad tiene mucho en común con el Sexo, todo el mundo lo quiere, todo el mundo cree que lo conoce, todo el mundo piensa que su ejecución sólo es cuestión de seguir las inclinaciones naturales y por supuesto, la mayoría de la gente piensa que los problemas en estas áreas están producidos por otra gente.

La garantía de Calidad del Software (SQA) es una "actividad de protección" que se aplica a lo largo de todo el proceso de Ingeniería Software.

La SQA engloba:

- Métodos y herramientas de análisis, diseño, codificación y prueba.
- Revisiones técnicas formales que se aplican durante cada paso de la Ingeniería del Software.
- Una estrategia de prueba multiescalada.
- El control de la documentación del Software y de los cambios realizados.
- Un procedimiento que asegure un ajuste a los estándares de desarrollo del Software (cuando sea posible).
- Mecanismos de medidas y de información.

Calidad del software y garantía de calidad del software

La calidad del software es la concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente.

Esta definición de calidad sirve para hacer hincapié en los siguientes puntos:

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.

Los estándares específicos 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.

Existe un conjunto de "requisitos implícitos" que a menudo no se mencionan. Por ejemplo: El deseo de un buen mantenimiento. Si el Software se ajusta a sus requisitos

explícitos pero falla en alcanzar los requisitos implícitos, la Calidad del Software 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.

Factores que determinan la calidad del software

Factores que pueden ser medidos directamente.

Por ejemplo: Errores/KLDC/unidad de tiempo.

Factores que sólo pueden ser medidos indirectamente.

Por ejemplo: Facilidad de uso o de mantenimiento.

En cualquiera de los dos casos se puede medir. Debemos comparar el Software con alguna referencia y llegar a una indicación de Calidad.

Clasificación de los Factores que afectan a la Calidad del Software, propuesta por McCall y sus colegas.

Los factores de Calidad del Software se centran en tres aspectos importantes de un producto de Software.

Sus características operativas.

- Facilidad de Mantenimiento (¿Puedo corregirlo?)
- Flexibilidad (¿Puedo cambiarlo?)
- Facilidad de prueba (¿Puedo probarlo?)

Su capacidad de soportar los cambios.

- Portabilidad (¿Podré usarlo en otra Máquina?)
- Reusabilidad (¿Podré reusar alguna parte del Software?)

Su adaptabilidad a nuevos entornos.

- Interoperabilidad (¿Podré hacerlo con otros Sistemas. ?)
- Corrección (¿Hace lo que quiero?)
- Fiabilidad (¿Lo hace de forma fiable todo el tiempo?)
- Eficiencia (¿Se ejecutará en mi hardware lo mejor que pueda?)
- Integridad (¿Es seguro?)
- Facilidad de uso (¿Está diseñado para ser usado?)

Descripción de los Factores de Calidad del Software, proporcionados por McCall.

***Corrección*:** Grado en que un programa satisface sus especificaciones y consigue los objetivos de la misión encomendada por el cliente.

Fiabilidad: Grado en que se puede esperar que un programa lleve a cabo sus funciones esperadas con la precisión requerida.

Eficiencia: Cantidad de recursos de computadora y de código requeridos por un programa para llevar a cabo sus funciones.

Integridad: Grado en que puede controlarse el acceso al Software o a los datos, por personal no autorizado.

Facilidad de uso: Esfuerzo requerido para aprender un programa, trabajar con él, preparar su entrada e interpretar su salida.

Facilidad de mantenimiento: Esfuerzo requerido para localizar y arreglar un error en un programa.

Flexibilidad: Esfuerzo requerido para modificar un programa operativo.

Facilidad de prueba: Esfuerzo requerido para probar un programa de forma que se asegure que realizara su función requerida.

Portabilidad: Esfuerzo requerido para transferir el programa desde un hardware y/o un entorno de Sistemas de Software a otro.

Reusabilidad: 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: Esfuerzo requerido para acoplar un sistema a otro.

Métricas usadas para desarrollar expresiones para cada uno de los factores de acuerdo con la siguiente relación:

$$Fc = c1 * m1 + c2 * m2 + \ldots + cn * mn$$

Fc: Es un factor de Calidad del Software.
ci: Son coeficientes de regresión.
mi: Son las métricas que afectan al factor de Calidad.

Desgraciadamente, muchas de las métricas definidas por McCall sólo pueden ser medidas de forma subjetiva. Las métricas pueden estar en forma de listas de comprobaciones usadas para " obtener el grado " de los atributos específicos del Software (CAV78). El esquema de graduación propuesto por McCall va en una escala de 0 (bajo) a 10 (alto).

Métricas usadas en el esquema de graduación.

Facilidad de auditoria: Facilidad con que se puede comprobar la conformidad con los estándares.

Exactitud: Precisión de los cálculos y del control.

Normalización de las comunicaciones: Grado en que se usan en ancho de banda, los protocolos y las interfaces estándar.

Completitud: Grado en que se ha conseguido la total implementación de las funciones requeridas.

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

Consistencia: Uso de un diseño uniforme y de técnicas de documentación a lo largo del proyecto de desarrollo del Software.

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

Tolerancia de errores: Daño que se produce cuando el programa encuentra un error.

Eficiencia en la ejecución: Rendimiento en tiempo de ejecución de un programa.

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

Generalidad: Amplitud de aplicación potencial de los componentes del programa.

Independencia del hardware: Grado en que el Software es independiente del hardware sobre el que opera.

Instrumentación: Grado en que el propio programa muestra su propio funcionamiento e indica errores que aparecen.

Modularidad: Independencia funcional de los componentes del programa.

Facilidad de operación: Facilidad de operación de un programa.

Seguridad: Disponibilidad de mecanismos que controlen o protejan los programas o los datos.

Autodocumentación: Grado en que el código fuente proporciona documentación significativa.

Simplicidad: Grado en que el programa puede ser entendido sin dificultad.

Independencia del sistema de Software: 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: Posibilidad de seguir la pista a la presentación del diseño o de los componentes reales del programa hacia atrás, hacia los requisitos.

Formación: Grado en que el Software ayuda para permitir que nuevos usuarios apliquen el sistema.

Se han obtenido libremente de trabajos anteriores, se pueden utilizar para establecer métricas de calidad en cada paso del proceso de Ingeniería del Software 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, la adaptabilidad y la utilidad (juntos representan la " 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.

Es un "planificado y sistemático diseño de acciones" que se requieren para asegurar la Calidad del Software. Lo que esto implica en el desarrollo de software es que la responsabilidad de la garantía del software corresponde muchos constituyentes de una organización (Ingenieros de software, gestores del proyecto, clientes, comerciales y personas que trabajan dentro del grupo de SQA.

La gente que lleva a cabo la SQA debe mirar el software desde el punto de vista del cliente intentando responder a las siguientes preguntas y otras cuestiones para asegurar que se mantiene la calidad del software.

1. ¿Satisface de forma adecuada el software los factores de calidad apuntados en la sección inicial?
2. ¿Se ha realizado el desarrollo del software de acuerdo con estándares preestablecidos?

3. ¿Han desempeñado apropiadamente sus papeles las disciplinas técnicas como parte de la actividad de SQA?

Actividades de SQA.

La garantía de calidad del software comprende una gran variedad de tareas, asociadas con siete actividades principales:

Aplicación de métodos técnicos: Ayudan al analista a conseguir una especificación de alta calidad y un diseño de alta calidad.

Realización de revisiones técnicas formales (RTF): Es una especie de reunión del personal técnico con el único propósito de descubrir problemas de calidad.

Prueba del software: Combina una estrategia de múltiples pasos con una serie de métodos de diseño de casos de prueba que ayudan a asegurar una efectiva detección de errores.

Ajuste a los estándares: El grado de aplicación de procedimientos y estándares en el proceso de la ingeniería del software varia 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. La garantía de seguimiento de estándares puede ser llevada a cabo por los encargados del desarrollo del software como parte de una revisión técnica formal o, en situaciones en que se requiera una verificación del seguimiento independiente, por el grupo SQA mediante su propia auditoria.

Control de cambios: Cada cambio realizado sobre el software en potencia puede introducir errores o crear efectos laterales que propaguen errores. El proceso de control de cambios es parte de la gestión de configuraciones del software y contribuye directamente a la calidad del software, al formalizar las peticiones de cambio, evaluar la naturaleza del cambio y controlar el impacto del cambio. Este se aplica durante el desarrollo del software y posteriormente durante la fase de mantenimiento del software.

Mediciones: Es una actividad integral para cualquier disciplina. Las métricas del software engloban un amplio conjunto de medidas técnicas orientadas a la gestión.

Registro y realización de informes: El registro de información y la generación de informes para la garantía de calidad del software dan procedimientos para la recolección y divulgación de información de SQA. Los resultados de las revisiones, auditorias, control de cambios, prueba y otras actividades de SQA 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.

Revisiones del software

Son un “filtro” para el proceso de ingeniería del software, se aplican en varios momentos del desarrollo del software y sirven para detectar defectos que puedan así ser eliminados, también para “purificar” las actividades de ingeniería del software que hemos denominado análisis, diseño y codificación.

Argumento sobre la necesidad de revisiones, por Freedman y Weinberg.

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 mas fácilmente al que los origina que a otras personas.

El proceso de revisión es, por tanto, la respuesta a la plegaria de Robert Burns: ¡Que gran regalo seria poder vernos como nos ven los demás!

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

- Señalar la necesidad de mejoras en el producto de una sola persona o un equipo.
- Confirmar las partes de un producto en las que no es necesaria o no es deseable una mejora.
- 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.

Impacto de los defectos del software sobre el coste

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.

Al detectar y eliminar un gran porcentaje de esos errores, el proceso de revisión reduce sustancialmente el coste de los siguientes pasos de las fases de desarrollo y mantenimiento.

7.2 MÉTRICAS DEL SOFTWARE

Las mediciones y las métricas nos ayudan a entender tanto el proceso técnico que se utiliza para desarrollar un producto, como el propio producto.

Frecuentemente en la medición surgen las siguientes interrogantes:

- ¿Cuáles son las métricas apropiadas para el proceso y para el producto?
- ¿Cómo se deben utilizar los datos que se recopilan?
- ¿Es bueno usar medidas para comparar gente, procesos o productos? entre otras.

Estimación

La *Planificación* es una de las actividades del proceso de Gestión de Proyectos de Software. Cuando se planifica un proyecto de Software se tiene que obtener

estimaciones: del esfuerzo humano requerido (personas-mes), de la duración cronológica del proyecto (en fechas) y del costo (dólares).

Las muchas técnicas del desarrollo del software tienen en común lo siguiente:

- Se ha de establecer de antemano el ámbito del proyecto
- Como base para la realización de estimaciones se usan las métricas del software (mediciones del pasado).
- El proyecto se desglosa en partes más pequeñas que se estiman individualmente

Análisis de Riesgos

El análisis de riesgo es algo vital para una buena gestión del proyecto de software y, sin embargo, a pesar de todo, se emprenden muchos proyectos sin que se hayan considerado los riesgos concretos.

El análisis de riesgo consiste realmente en una serie de pasos de control de los riesgos tales como:

- Identificación de riesgos
- Solución de riesgos
- Supervisión de riesgos, etc.

Planificación Temporal

La planificación de un proyecto de software no difiere de la planificación de cualquier proyecto de ingeniería por consiguiente:

- Se identifica una serie de tareas del proyecto
- Se establecen interdependencias entre las tareas
- Se estima el esfuerzo asociado con cada tarea
- Se hace la asignación de personal y de otros recursos
- Se crea una “red de tareas”
- Se desarrolla una agenda de fechas.

Seguimiento y Control

Si una tarea se sale de la agenda el gestor puede utilizar una herramienta de planificación automática sobre el proyecto para determinar el impacto del error de planificación sobre los hitos intermedios y sobre la fecha final de entrega. De manera que se pueden reasignar los recursos, reordenar las tareas o (como último recurso) modificar los compromisos de entrega para resolver el problema no detectado.

De este modo se puede controlar mejor el desarrollo del software.

Métrica para la productividad y calidad del software

La medición es fundamental para cualquier disciplina de ingeniería. Las métricas del software se refieren a un amplio rango de medidas para el software de computadoras.

Medición del software

La medición se aleja de lo común en el mundo de la ingeniería de software sin embargo existen varias razones para medirlo:

- Para indicar la calidad del producto.
- Para medir la calidad de la gente que desarrolla el producto.
- Para evaluar los beneficios (en términos de productividad y calidad) derivados del uso de nuevos métodos y herramientas de ingeniería de software.
- Para establecer una línea de base para la estimación.
- Para ayudar a justificar el uso de nuevas herramientas de formación adiciona.

Las métricas del software se pueden clasificar en:

1. **Métricas de productividad**: Se centran en el rendimiento de procesos de ingeniería de software.
2. **Métricas de calidad:** Conveniencia del software para su utilización.
3. **Métricas técnicas:** Se centran en las características del software.

También se puede hacer una segunda clasificación.

1. Métricas orientadas al tamaño.
2. Métricas orientadas a la función.
3. Métricas orientadas a la persona.

Métricas orientadas al tamaño.

Estas son medidas directas del software y el proceso por el cual se desarrolla.

Ecuaciones para calcular la productividad y calidad del software.

PRODUCTIVIDAD = KLDC / PERSONAS-MES
CALIDAD = ERRORES / KLDC

Además se pueden calcular otras métricas interesantes.

COSTE = DÓLARES / KLDC
DOCUMENTACIÓN = PAG. DE DOC./ KLCD

KLDC : Miles de líneas de códigos.

Cinco ejemplos de métricas del proceso

1. Número de defectos por KLOC detectados dentro de 12 semanas de la entrega
2. Varianza en cada etapa del programa

$$\frac{duracionreal - duracionproyectada}{duracionproyectada}$$

3. Varianza en el costo

$$\frac{\cos toreal - \cos toproyectado}{\cos toproyectado}$$

4. Tiempo de diseño total/tiempo de programación total
    - debe ser al menos 50%
5. Tasas de inyección y detección de defectos por etapas
    - por ejemplo, “un defecto por clase en la etapa de diseño detallada”

Métricas orientadas a la función.

Son medidas indirectas el software y el proceso por el cual se desarrolla. Las métricas orientadas a función fueron propuestas por Albrecht quien sugirió los puntos de función, estos se obtienen utilizando una evaluación empírica basada en medidas cuantitativas y valoraciones subjetivas de complejidad del software.

Los valores del ámbito de la información están definidos de la siguiente manera:

- Número de entrada del usuario
- Número de salida del usuario
- Número de peticiones del usuario
- Número de archivos
- Número de interfaces externas

Para calcular los puntos de función, se utiliza la siguiente relación:

PF = cuenta-total x [0.65 + 0.01 x SUM (F_i)]

Cálculo de métricas de Puntos de Función

Parámetro de medida	Cuenta	Factor de Peso Simple	Medio	Complejo	
Número de entradas de usuario		x 3	4	6 =	
Número de salidas de usuario		x 4	5	7 =	
Número de peticiones al usuario		x 3	4	6 =	
Número de archivos		x 7	10	15 =	
Número de Interfaces externas		x 5	7	10 =	
Cuenta_Total					

F_i son los valores de ajuste de complejidad basados en las respuestas a las cuestiones siguientes:

Evaluar cada factor en una escala de 0 a 5:

0	1	2	3	4	5
Sin Influencia	Incidental	Moderado	Medio	Significativo	Esencial

1. ¿Requiere el sistema copias de seguridad y de recuperación confiables?
2. ¿Se requieren comunicaciones de datos?
3. ¿Existen funciones de procesamiento distribuido?
4. ¿Es crítico el rendimiento?
5. ¿Será ejecutado el sistema en un entorno operativo existente y fuertemente utilizado?
6. ¿Requiere el sistema entrada de datos interactiva?
7. ¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples pantallas o variadas opciones?
8. ¿Se actualizan los archivos maestros de forma interactiva?
9. ¿Son complejas las entradas, las salidas, los archivos o las peticiones?
10. ¿Es complejo el procesamiento interno?
11. ¿Se ha diseñado el código para ser reutilizable?
12. ¿Están incluidas en el diseño la conversión y la instalación?
13. ¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?
14. ¿Se ha diseñado la aplicación para facilitar los cambios y para ser fácilmente utilizada por el usuario?

Una vez calculado los Puntos de Función, se usan de forma análoga a las LDC como medida de la productividad, calidad y otros atributos del software:

- **Productividad =** PF / persona-mes
- **Calidad =** Errores / PF
- **Costo =** Dólares / PF
- **Documentación =** Paginas de documentación / PF, entre otras.

La medida de PF se diseñó originalmente para ser utilizada en aplicaciones de sistemas de información de gestión. Sin embargo las ampliaciones propuestas por Jones, en las que los denomina p*untos de característica*, pueden permitir que se utilice esta medida en aplicaciones de software de ingeniería y de sistemas. La métrica del Punto de Característica tiene en cuenta otra característica del software denominada *los algoritmos* que se definen como “un problema de complejidad computacional limitada que se incluye dentro de un determinado programa de computadora”.

Para calcular el Punto de Característica se utiliza la siguiente tabla.

Parámetro de medida	Cuenta	Peso		
Número de entradas de usuario		x 4	=	
Número de salidas de usuario		x 5	=	
Número de peticiones al usuario		x 4	=	
Número de archivos		x 7	=	
Número de Interfaces externas		x 7	=	
Algoritmos		x 3	=	
Cuenta_Total				

Métrica para la Calidad del Software

Podemos medir la calidad a lo largo del proceso de Ingeniería el Software y una vez que el software se ha distribuido al cliente y a los usuarios.

Las métricas de calidad de esta categoría incluyen:

1. La complejidad del programa
2. La modularidad efectiva
3. El tamaño del programa

Visión General de los Factores que afectan a la Calidad

McCall y Cavano definieron un conjunto de factores de calidad los cuales evaluaban al software desde 3 puntos de vista distintos:

1. Operación del producto (Su uso)
2. Revisión del producto (Su modificación)
3. Transición del producto (Transportarlo)

Medidas de Calidad

Según Gilb existen muchas formas de medir la calidad del software, dentro de las más utilizadas tenemos las métricas *a posteriori*:

Corrección Es el estado con que el software realiza la función requerida. La medida más común es el número de defectos por KLDC, donde *defecto* se define como carencia verificada de conformidad con los requisitos.

Facilidad de Mantenimiento Es la facilidad con la que se puede corregir un programa si se encuentra un error; adaptarlo si su entorno cambia o mejorarlo si el cliente desea un cambio en los requisitos.

Integridad Para medir la integridad se tiene que definir dos atributos:

- *Amenaza*: Es la probabilidad (que se pude estimar o deducir de la evidencia empírica) de que un ataque de un tipo determinado ocurra en un tiempo determinado.
- *Seguridad:* Es la probabilidad (que se pude estimar o deducir de la evidencia empírica) de que se pueda repeler el ataque de un determinado tipo.

La Integridad del sistema puede definirse como:

Integridad = [1- Amenaza x (1 – Seguridad)]

Reconciliación de las diferentes métricas

La relación entre la línea de código y la línea de función dependen del lenguaje de programación que se utilice para implementar el software y la calidad del diseño.

Ciertos estudios han intentado relacionar LDC y PF. La siguiente lista proporciona estimaciones informales del número medio de líneas de código requerido para construir un Punto de Función en varios lenguajes de programación.

Lenguaje de Programación	LDC/PF (media)
Ensamblador	300
Cobol	100
Fortran	100
Pascal	90
ADA	70
Lenguaje orientado a los objetos	30
Lenguaje de IV generación (L4G)	20
Generadores de código	15

Basili y Zelkowitz definen 5 factores importantes que inciden en la productividad del software:

Factores humano: El tamaño y la experiencia de la organización del desarrollo.

Factores del problema: La complejidad del problema a resolver y el número de cambios en las restricciones o los requisitos del diseño.

Factores del proceso: Técnicas de análisis y diseño utilizadas, disponibilidad del lenguaje y herramientas CASE y de técnica de revisión y de técnica de revisión.

Factores del producto: Fiabilidad y rendimiento basados en computadoras.

Factores de los recursos: Disponibilidad de herramientas CASE, de recurso de Hardware y Software.

Integración de las métricas dentro del proceso de la Ingeniería del Software

En esta sección consideraremos algunos argumentos sobre la métrica del software.

¿Por qué es tan importante medir el proceso de Ingeniería del Software?

- Permite determinar si estamos mejorando
- Proporciona beneficio a nivel estratégico
- Proporciona beneficio a nivel de proyecto
- Proporciona beneficio a nivel técnico

Mediante la obtención y la evaluación de medidas de calidad y de productividad, los directivos de gestión pueden establecer objetivos significativos para mejorar el proceso de ingeniería de software.

Establecimiento de una Línea Base

Para que sea una ayuda efectiva en la planificación estratégica y/o en las estimaciones de costo y esfuerzos, los datos de la Línea Base tienen que poseer los siguientes atributos:

- Los datos deben ser razonablemente precisos, han de evitarse las "suposiciones" sobre proyectos anteriores.
- Los datos deben obtenerse de tantos proyectos como sea posible.
- Los medios tienen que ser consistentes.
- Las aplicaciones deben ser similares a la que vaya a ser estimada.

Recolección, Cálculo y Evaluación de Métricas.

Proceso de Ingeniería del Software				Gestores
	Recolección de Datos		Evaluación de Datos	
Software		Cálculo de Métricas		Desarrolladores

Gráfico. Proceso de Recolección de las Métricas del Software

Recolección de Datos:
Requiere una investigación histórica de proyectos pasados para reconstruir los datos requeridos.

Calculo de Métrica: Las métricas pueden abarcar un amplio rango de medida de LCD ó PF.

Evaluación de Datos: Se centran en las razones intrínsecas de los resultados obtenidos.

La gestión del proyecto de software representa el primer nivel del proceso de ingeniería del software. La gestión del proyecto está compuesta por actividades que incluyen la medición, estimación, análisis de riesgos, planificación, seguimiento y control.

La medición permite a los gestores y desarrolladores entender mejor el proceso de ingeniería del software y el producto que se produce.

La medición produce un cambio cultural. La obtención de datos, el cálculo de métricas y la evaluación de datos son los 3 pasos que se tienen que implementar para andar a echar un programa de métricas. Mediante la creación de una línea base los ingenieros y sus gestores pueden ganar un mayor rendimiento sobre el trabajo que realizan y sobre el producto que generan.

7.3 LAS PRUEBAS DEL SOFTWARE

En término generales, las pruebas se definen como el proceso de localizar y reparar los errores. La fase de prueba en el desarrollo del software es vista como la actividad anterior a la presentación del producto final.

Dado que una prueba completa no es viable, la pregunta de que tantas pruebas y en donde deben hacerse depende en un alto grado de la aplicación en particular, por ejemplo en un simple paquete de procesador de palabras no requiere de muchas pruebas rigurosas, caso contrario sería en un sistema complejo como los utilizados en las áreas militares, en investigaciones científicas, en los hospitales, etc.

Objetivo

El objetivo de una prueba es de lograr un producto de calidad, un producto que cumpla con las especificaciones del software, un producto en la cual el cliente quede satisfecho.

Importancia

Las pruebas son de gran importancia, ya que de ellas depende de que el producto sea aceptado o no, una prueba será exitosa si se logran localizar las fallas, en caso contrario tendrá un costo muy alto sobre todo en sistemas que están en producción.

Se puede realizar las siguientes pruebas:

Pruebas al azar: Esta prueba involucra tanto al programador o a un probador, intentando hacer fallar al sistema, ya sea a nivel de módulo, pero es casi siempre a

nivel del sistema, la estrategia que Se utiliza es suprimiendo campos que no existen o introduciendo entradas incorrectas.

Prueba manual: Esta prueba consiste en realizar todos los casos de pruebas a mano, luego son introducida en la máquina manualmente y Se verifican los resultados con los realizados a mano a ver si concuerdan con las salidas esperadas.

Programadores que prueban su propio código: Este tipo de método no es recomendable, aunque sea el programador el que mejor conozca su código.

Principios de la prueba
Antes de la aplicación de métodos para el diseño de casos de prueba efectivos, un ingeniero del software deberá entender los principios básicos que guían las pruebas del software. Por ejemplo:

- A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente
- Las pruebas debería planificarse mucho antes de que empiecen
- El principio de Pareto es aplicable a la prueba del software
- La pruebas deberían empezar por “lo pequeño” y progresar hacia “lo grande”
- No son posibles las pruebas exhaustivas
- Para ser más efectivas, las pruebas deberían ser conducidas por un equipo independiente

Facilidad de prueba
En circunstancias ideales, un ingeniero del software diseña un programa de computadora, un sistema, o un producto con la “facilidad de prueba” en mente. Esto permite a los encargados de las pruebas diseñar casos de pruebas más fácilmente. Pero, ¿qué es la facilidad de prueba? James Bach describe la facilidad de prueba de la siguiente manera:

La facilidad de prueba del software es simplemente lo fácil que se puede probar un programa de computadoras. Como la prueba es tan profundamente difícil, merece la pena saber qué se puede hacer para hacerlo más sencillo.

Existen, de hecho, métricas que podrían usarse para medir la facilidad de prueba en la mayoría de sus aspectos. A veces, la facilidad de prueba se usa para indicar lo adecuadamente que un conjunto particular de pruebas va a cubrir un producto.

La siguiente lista de comprobación proporciona un conjunto de características que llevan a un software fácil de probar.

- Operatividad.- Cuanto mejor funcione, más eficientemente se puede probar
- Observabilidad.- Lo que ves es lo que pruebas
- Controlabilidad.- Cuanto mejor podamos controlar el software, más se puede automatizar y optimizar
- Capacidad de descomposición.- Controlando el ámbito de las pruebas podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión
- Simplicidad.- Cuanto menos haya que probar, más rápidamente podemos probarlo
- Estabilidad.- cuanto menos cambios, menos interrupciones a las pruebas

- Facilidad de comprensión.- Cuanta más información tengamos, más inteligente serán las pruebas

Diseño de casos de pruebas

Cualquier producto de ingeniería (y de muchos otros campos) pueden comprobarse de una de estas dos formas:

1. Conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa, y al mismo tiempo buscando errores en cada función. A esta prueba se le denomina prueba de caja negra
2. Conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que todas las piezas encajan, o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. A esta prueba se le denomina prueba de caja blanca.

PRUEBA DE CAJA BLANCA

A veces se le denomina prueba de caja de cristal y se basa en un minucioso examen de los detalles procedimentales. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de pruebas que:
1. garanticen que se ejecute por lo menos una vez todos los caminos independientes de cada módulo
2. se ejecuten todas las decisiones lógicas en sus vertientes verdadera y falsa
3. se ejecuten todos los bucles en sus límites y con sus límites operacionales, y
4. se ejecuten las estructuras internas de datos para asegurar su validez

Prueba del camino básico

Es una técnica de prueba de caja blanca propuesta inicialmente por Tom McCabe. El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.

- Notación de grafo de flujo.- Antes de considerar el método del camino básico se debe introducir una sencilla notación para la representación del flujo de control, denominada grafo de flujo (o grafo del programa). El grafo de flujo representa el flujo de control lógico
- Complejidad ciclomática.- Es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba del camino básico, el valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecutan cada sentencia al menos una vez
- Obtención de casos de prueba.- El método de prueba de camino básico se puede aplicar a un diseño procedimental detallado o a un código fuente
- Matrices de grafos.- El procedimiento para obtener el grafo de flujo e incluso la determinación de un conjunto de caminos básicos es susceptible de ser mecanizado. Para desarrollar una herramienta de software que ayude en la prueba del camino básico, puede ser bastante útil una estructura de datos denominada

matriz de grafo. Una matriz de grafos es una matriz cuadrada cuyo tamaño (el número de filas y de columnas) es igual al número de nodos del grafo del flujo. Cada fila y cada columna corresponde a un nodo específico y las entradas de la matriz corresponde a las conexiones (aristas) entre los nodos.

Prueba de la estructura de control

La técnica de prueba del camino básico es una de las muchas técnicas para la prueba de la estructura de control. Aunque la prueba del camino básico es sencilla y altamente efectiva, no es suficiente por sí sola, a continuación se presentaran algunas variantes

- Prueba de condición.- es un método de diseño de caos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa. Una condición simple es una variable lógica o una expresión relacional, posiblemente precedida con un operador NOT.

  Si una condición es incorrecta, entonces es incorrecto al menos un componente de la condición. Así, los tipos de errores de una condición pueden ser los siguientes:

  - ◊ Error en el operador lógico (existencia de operadores lógicos incorrectos / desaparecidos / sobrantes)
  - ◊ Error en variable lógica
  - ◊ Error en paréntesis lógico
  - ◊ Error en operador relacional
  - ◊ Error en expresión aritmética

- Prueba de flujo de datos.- el método de prueba de flujo de datos selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables el programa
- Pruebas de bucles.- Los bucles son las piedras angulares de la inmensa mayoría de los algoritmos implementados en software. Y, sin embargo, les prestamos normalmente poca atención cuando llevamos a cabo la prueba del software. La prueba de bucles es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles. Se pueden definir cuatro clases de diferentes de bucles:

  - ◊ Bucles simples.- A los bucles simples se les debe aplicar al siguiente conjunto de pruebas, donde n es el número máximo de pasos permitidos por el bucles:
    - Pasar por alto totalmente el bucle
    - Pasar una sola vez por el bucle
    - Pasar dos veces por el bucle
    - Hacer m pasos por el bucle con m<n
    - Hacer n - 1, n y n + 1 pasos por el bucle
  - ◊ Bucles anidados.- Beizer sugiere el siguiente enfoque
    1. Comenzar por el bucle más interior. Establecer o configurar los demás bucles con sus valores mínimos
    2. Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras se mantienen los parámetros de iteración de los bucles externos en sus valores mínimos.

3. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores “típicos”
4. Continuar hasta que se hayan probado todos los bucles

◊ Bucles concatenados.- Se pueden probar mediante el enfoque anteriormente definido par los bucles simples, mientras cada uno de los bucles sea independiente del resto. Sin embargo, si hay dos bucles concatenados y se usa el controlador del bucle 1 como el valor inicial del bucle 2, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados

◊ Bucles no estructurados.- Siempre que sea posible, esta clase de bucles se deben rediseñar para que se ajusten a las construcciones de programación estructurada

PRUEBA DE CAJA NEGRA

Las pruebas de caja negra se centran en los requisitos funcionales del software. O sea, la prueba de caja negra permite al ingeniero del software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. La prueba de caja negra no es una alternativa a las técnicas de prueba de caja blanca. Más bien se trata de un enfoque complementario.

La prueba de caja negra intenta encontrar errores de las siguientes categorías:
1. funciones incorrectas o ausentes
2. errores de interfaz
3. errores en estructuras de datos o en accesos a bases de datos externas
4. errores de rendimiento y
5. errores de inicialización y terminación

Métodos de prueba basada en grafos

El primer paso en la prueba de caja negra es entender los objetos que se modelan en el software y las relaciones que conectan a estos objetos. Una vez que se ha llevado a cabo esto, el siguiente paso es definir una serie de pruebas que verifique que todos los objetos tienen entre ellos las relaciones esperadas.

Para llevar a cabo estos pasos, el ingeniero del software empieza creando un grafo (una colección de nodos que representan objetos); enlaces que representan la relaciones entre los objetos; pesos de nodos que describen las propiedades de un nodo y pesos de enlaces que describen alguna característica de un enlace.

Equivalencia de Partición

Esta es una técnica de prueba de caja negra que se refiere a identificar clases de entradas sobre las cuales el comportamiento del sistema es similar. Por ejemplo, si un programa produce una cierta salida cuando un entero entre 1 y 100 se introduce, y diferentes salidas dependiendo de si el entero es menor que 1, de 1 a 100, y mayores que 100. La idea de particiones equivalentes es que las entradas en las clases sean representativas de la clase como un todo y que la cantidad de datos de prueba requeridos pueda, por lo tanto, ser minimizada.

Análisis de valores límites

Esta técnica es un paso adelante de la partición equivalente con casos de prueba que son seleccionados de clases separadas. La técnica explora los valores límites de las clases de entrada. Por ejemplo, considere un modulo que requiere un entero en el rango de 1 a 20. Los casos de prueba útiles serian 1,0, -1,19,20,21, -20.Otros casos de prueba pudieran ser un entero grande, un decimal, un entero muy pequeño y así sucesivamente.

Prueba de Unidad

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software: el módulo. Usando la descripción del diseño procedimental como guía, se prueban los caminos de control importantes, con el fin de descubrir errores dentro del límite del módulo.

La prueba de unidad siempre esta orientada a caja blanca y este se puede llevar a cabo en paralelo para múltiples módulos.

Consideraciones sobre la prueba de unidad

Se prueba la *interfaz* del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa que está siendo probada. Se examinan las *estructuras de datos locales* para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las *condiciones de límite* para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones e procedimiento. Se ejercitan todos los *caminos independientes* (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. Y, finalmente, se prueban todos los *caminos de manejos de errores.*

Prueba de Integración

La prueba de integración es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados son la interacción. El objetivo es coger los módulos probados en unidades y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.

A menudo hay una tendencia a intentar una integración no incremental; es decir, a construir el programa mediante un enfoque de “big bang”. Se combinan todos los módulos por anticipado. Se prueba todo el programa en conjunto. Normalmente se llega al caos!. Se encuentra un gran conjunto de errores. La corrección se hace difícil, puesto que es complicado aislar las causas al tener delante el programa entero en toda su extensión. Una vez que se corrigen esos errores aparecen otros nuevos y el proceso continúa en lo que parece ser un ciclo sin fin.

La integración incremental es la antítesis del enfoque “big bang”. El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir, es más probable que se puedan probar completamente las interfaces y se puede aplicar un enfoque de prueba sistemática.

Integración Descendente

Es un planteamiento incremental a la construcción de la estructura de programas. Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal (programa principal). Los módulos subordinados (de cualquier modo subordinados) al módulo de control principal se van incorporando en la estructura, bien de forma primero en profundidad, o bien de forma primero en anchura.
El proceso de integración se realiza en una serie de cinco pasos:

1. Se usa el módulo de control principal como controlador de la prueba, disponiendo de resguardos para todos los módulos directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración elegido (es decir, primero en profundidad o primero en anchura) se van sustituyendo los resguardos subordinados uno a uno por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo
4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real
5. Se hace la prueba de regresión para asegurar de que no se han introducido errores nuevos.

Integración ascendente

Como su nombre lo indica, empieza la construcción y la prueba con los módulos atómicos (es decir, módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el proceso requerido de los módulos subordinados a un nivel dado siempre está disponible y se elimina la necesidad de resguardos.

Se puede implementar una estrategia de integración ascendente mediante los siguientes pasos:

1. Se combina los módulos de bajo nivel en grupos (a veces denominados construcciones) que realicen una subfunción específica del software.
2. Se escribe un controlador (un programa de control de prueba) para coordinar la entrada y la salida de los casos de prueba
3. Se prueba el grupo
4. Se eliminan los controladores y se combina los grupos moviéndose hacia arriba por la estructura del programa

Prueba de regresión

Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el software cambia. Se establece nuevos caminos de flujo de datos, pueden ocurrir nuevas E/S y se invoca una nueva lógica de control.

Estos cambios pueden causar problemas con funciones que antes trabajaban perfectamente. En el contexto de una estrategia de prueba de integración, la prueba de regresión es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos

colaterales no deseados.. La prueba de regresión es la actividad que ayuda a asegurar que los cambios (debidos a las pruebas o por otros motivos) no introducen un comportamiento no deseado o errores adicionales.

La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas de reproducción de captura. Las herramientas de reproducción de captura permiten al ingeniero del software capturar casos de prueba y los resultados para la subsiguiente reproducción y comparación. El conjunto de pruebas de regresión contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de prueba que ejecute todas las funciones del software
- Pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio
- Pruebas que se centran en los componentes del software que se han cambiado

Prueba alfa y beta

La mayoría de los constructores de software llevan a cabo un proceso denominado prueba alfa y beta para descubrir errores que parezca que sólo el usuario final puede descubrir.

La prueba alfa se lleva a cabo en el lugar de desarrollo pero por un cliente. Se usa el software de forma natural con el desarrollador como observador del usuario y registrando los errores y los problemas de uso. Las pruebas alfa se llevan a cabo en un entorno controlado.

La prueba beta se lleva a cabo por los usuarios finales del software en los lugares de trabajo de los clientes. A diferencia de la prueba alfa, el desarrollador no está presente normalmente. Así, la prueba beta es una aplicación “en vivo” del software en un entorno que no puede ser controlado por el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que encuentra durante la prueba beta e informa a intervalos regulares al desarrollador. Como resultado de los problemas informados durante la prueba beta, el desarrollador del software lleva a cabo modificaciones y así prepara una versión del producto de software para toda clase de clientes.

PRUEBA DEL SISTEMA

La prueba del sistema, realmente, está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejecutar profundamente el sistema basado en computadora. Aunque cada prueba tiene un propósito diferente, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

Prueba de recuperación

Muchos sistemas basados en computadoras deben recuperarse de los fallos y continuar el proceso en un tiempo previamente especificado. En algunos casos, un sistema debe ser tolerante con los fallos; es decir, los fallos del proceso no deben hacer que cese el funcionamiento de todo el sistema

La prueba de recuperación es una prueba del sistema que fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente. Si la recuperación es automática (llevada a cabo por el propio sistema) hay que evaluar la corrección de la inicialización, de los mecanismos de recuperación del estado del sistema, de la recuperación del estado del sistema, de la recuperación de datos y del proceso rearranque. Si la recuperación requiere la intervención humana, hay que evaluar los tiempos medios de reparación para determinar si están dentro e unos límites aceptables

Prueba de seguridad

La prueba de seguridad intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho, de accesos impropios.

Durante la prueba de seguridad, el responsable de la prueba desempeña el papel de un individuo que desea entrar en el sistema. Debe intentar conseguir las claves de acceso por cualquier medio, puede atacar al sistema con software a medida, diseñado para romper cualquier defensa que se hay construido, debe loquear el sistema, negando así el servicio a otras personas, debe producir a propósito errores del sistema, intentando acceder durante la recuperación o debe curiosear en los datos sin protección, intentando encontrar la clave de acceso al sistema, etc.

El papel del diseñador del sistema es hacer que el costo de la entrada ilegal sea mayor que el valor de la información obtenida.

Prueba de resistencia
Las pruebas de resistencia están diseñadas para enfrentar a los programas con situaciones anormales. En esencia, el sujeto que realiza la prueba de resistencia se pregunta: ¿A qué potencia puedo ponerlo a funcionar antes de que falle?

La prueba de resistencia ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. Por ejemplo:

1. diseñar pruebas especiales que generen diez interrupciones por segundo, cuando las normales son una o dos:
2. incrementar las frecuencias de datos de entrada en un orden de magnitud con el fin de comprobar cómo responden las funciones de entrada;
3. ejecutar casos de prueba que requieran el máximo de memoria o de otros recursos;
4. diseñar caos de prueba que puedan dar problemas en un sistema operativo virtual o
5. diseñar casos de prueba que produzcan excesivas búsquedas de datos residentes en disco

Prueba de rendimiento

Para sistemas de tiempo real y sistemas empotrados, es inaceptable el software que proporciona las funciones requeridas pero no se ajusta a los requisitos de rendimiento. La prueba de rendimiento está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. La prueba de rendimiento se da durante todos los pasos del proceso de la prueba.

7.4 CONFIGURACIÓN Y MANTENIMIENTO DEL SOFTWARE

Es posible decir que nadie sabe con exactitud cuánto gasta una organización grande al año en mantenimiento de software. Sin embargo, hay un cuerpo considerable de información acumulada en esta área y podemos hacer un supuesto con bases. Los estudios tomados y publicados indican que entre el 40 y 70 % de todo el gasto de software es para el mantenimiento. Con el crecimiento anticipado de la producción del software ilustramos la necesidad de asegurar qué tanto cuidado se pone en el mantenimiento del software como en su creación.

Este tema proporciona una panorámica del proceso de mantenimiento de software. Sirve para establecer el fondo del asunto y para destacar las áreas críticas concernientes. Además de revisar el conocimiento existente, se presenta una visión de lo que está a la vanguardia del mantenimiento.

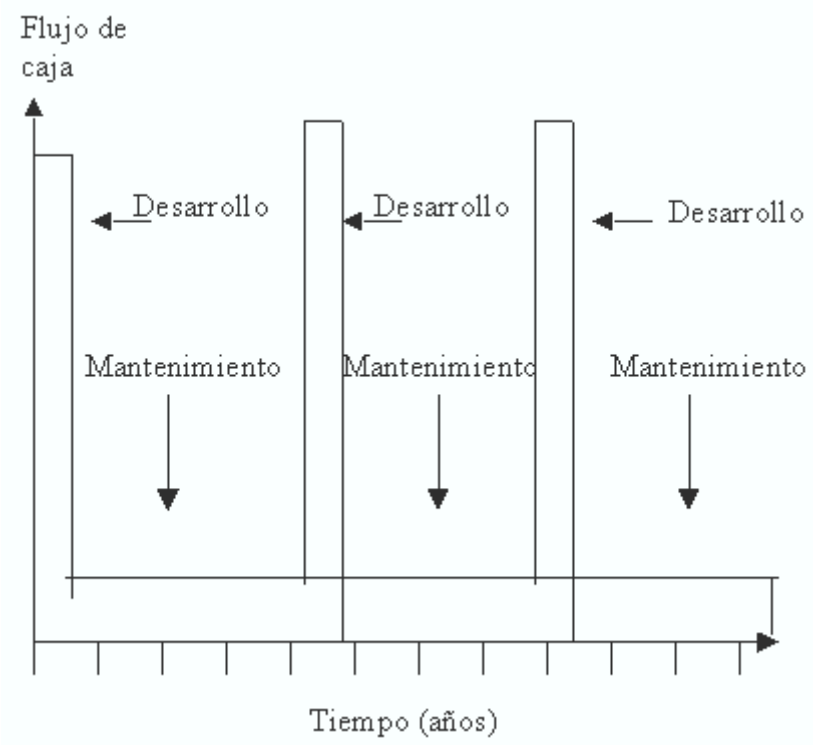
¿Qué significa Mantenimiento de Software?

Es la primera fuente de confusión porque diferentes autores y practicantes tienen distintos puntos de vista. No es mucho lo que difiere sobre lo que tiene que hacerse; si no lo que argumentan cada uno.

La parte del mantenimiento se puede tratar de tres formas:

a. Como una simple actividad al final: Aunque irrefutable conviene poco entendimiento de lo que ocurre en la actualidad.
b. En diagramas: Indica de manera correcta que la especificación, el diseño, etc. Son de hecho parte de la actividad del mantenimiento.
c. Ignorándola.

En la siguiente figura se ilustra un ciclo de vida de mantenimiento simple pero útil. Este muestra períodos de intensa actividad (flujo de caja alto) cuando los cambios se hacen a un producto ya existente. Intercalado con esto se encuentran los períodos de calma (flujo de caja bajo) cuando los únicos cambios que se hacen al producto son para la corrección de errores.



Definición de Mantenimiento de Software

Es el conjunto de actividades asociadas con el hecho de mantener un software operacional a tono con los requerimientos de sus usuarios y operadores y de toda la gente y sistemas con los que el sistema operacional interactúa.

Se consideran cuatro categorías de mantenimiento:

Mantenimiento Correctivo (mantenimiento de reparación): Involucra la corrección de errores en una parte del software donde un error se define como una desviación de la especificación. Predecir la extensión del mantenimiento correctivo que cualquier programa nuevo requiere, es cuestión de interés y por lo tanto, progreso.

Mantenimiento Adaptativo: Involucra la adaptación de un programa para traerlo a línea con los cambios en su especificación. Estos cambios pueden resultar de nuevos requerimientos de usuario o de un cambio en el ambiente del sistema operacional.

Mantenimiento Perfectivo (mantenimiento productivo): Este no altera ni la especificación ni la adherencia del software a él mismo, pero mejora el desempeño al hacer que el software consuma menos recursos, la palabra “recursos” incluye factores como tiempo de ejecución y uso de memoria.

Mantenimiento Preventivo: Esta categoría final dirige sus beneficios hacia el proceso general del mantenimiento mismo. Implica hacer cambios al software que por sí mismo, no mejora ni la corrección ni el desempeño, pero provocan que las actividades futuras de mantenimiento sean más fáciles (baratas de llevar a cabo).

A estas definiciones podemos agregar:

Mantenibilidad: La facilidad con la que un sistema de software puede corregirse cuando ocurren errores o deficiencias, y pueden ser extendido o comprimido para satisfacer nuevos requerimientos.

Existen muchas razones por las que el mantenimiento del software es difícil, algunas inherentes, otras debido a la falta de investigación, desarrollo y medición objetiva de los procesos mismos.

De manera básica, los programadores de mantenimiento deben enfrentarse con grandes y aún crecientes volúmenes de software, todos son usados para tratar con sistemas cuya documentación es inapropiada para sus necesidades o se carece de ella. Esto hace que el mantenimiento sea costoso. Puesto de otra forma, el mantenimiento es difícil en un sistema que no ha sido diseñado para recibir mantenimiento.

Las fuentes específicas de dificultad son:

- Del 75 al 80% del software existente fue producido previo al uso significativo de la programación estructurada.
- Es difícil relacionar aspectos particulares del comportamiento de un sistema con un código específico.
- Los cambios a menudo no están documentados de manera adecuada.
- Hay un efecto de ondulación; por lo general, los cambios a una parte del código no se localizan.
- Todos estos factores son tan difíciles que solo se pueden facilitar con técnicas más sofisticadas para el mantenimiento.

El mantenimiento en el ciclo de vida del software

En el mantenimiento del ciclo de vida del software, se desprende la existencia de cuatro fases fácilmente identificables por las que un producto liberado atraviesa:

Fase de Realce: La especificación del producto lleva una alteración considerable. Durante esta fase, al menos una versión inicial del producto está en uso y de manera simultánea, se hacen grandes cambios en la preparación de la siguiente versión planeada.

El problema de recursos es difícil, y una forma de redondearla es instalar un equipo separado para manejar los cambios al producto “viejo”. Esto significa la administración y el control de recursos, pero tiene tres desventajas:

- La comunicación entre las dos partes se vuelve más distante.
- Hay duplicación de esfuerzo.
- Hay un problema de autoridad.

Fase de Madurez: La actividad principal es el mantenimiento correctivo. Quizá el problema más significativo de esta fase, es el deterioro del sistema. Cada cambio hecho al software corre algún riesgo de introducir un nuevo error y, si el riesgo no puede mantenerse dentro de sus límites, el desempeño actual del producto se deteriorará con el tiempo en lugar de mejorar.

Fase de Obsolescencia: Esta fase es la precursora de la muerte técnica, o sea que el producto sigue disponible y en uso pero la habilidad para modificarlo se ha perdido. El principio de esta fase puede ser provocado por varios aspectos:

- La pérdida de la única persona que entiende un programa no documentado.
- La falta de habilidad para mantener el soporte de software o hardware.
- Las pérdidas inadvertidas del código fuente.
- Decisión deliberada y racional.

Fase de Terminación: En la que el producto puede continuar en uso pero todo el control ha sido abandonado.

La administración del mantenimiento

Esta actividad, involucra la creación y preservación de un ambiente que permitirá que se lleve a cabo la administración del mantenimiento de la mejor manera.

Se consideran las siguientes áreas:

1. **Planeación y Monitoreo**: La planeación del mantenimiento lleva consigo el problema particular de que es muy difícil estimar la demanda probable de trabajo, puesto que nadie sabe de antemano cuántos errores se descubrirán en software y el número de cambios que se harán en los requerimientos de los usuarios .

   La planeación no es una actividad de una sola vez en el mantenimiento: Las proyecciones deberán ser revisadas varias veces durante la vida de un producto. Debe, al menos en teoría volverse más fácil con el tiempo puesto que será experiencia a la cual recurrir. Sin embargo la única forma de saberlo es mantener los registros: monitorear la actividad.

2. **El Problema de la Entrega**: Es importante tener un conjunto claro de normas para la entrega; por ejemplo, la documentación proporcionada por el equipo de diseño debe ayudar a los encargados del mantenimiento a entender el software, y este debe incluir facilidades apropiadas para construir, probar y administrar los archivos que constituyen el sistema.

3. **Entrenamiento**: El encargado de un mantenimiento con éxito necesita saber “todo sobre todo”. Las áreas más importantes en las cuales es necesario

asegurar la competencia técnica son la administración de la configuración y las técnicas de análisis de código, ya que la base del mantenimiento es la localización y revisión ordenada del código existente.

4. **Control de Recursos**: Este es un problema de administración general, pero hay dos aspectos que se relacionan para el mantenimiento:

   a) En una organización, ¿dónde debe llevarse a cabo el mantenimiento?, ¿Debe hacerlo la misma gente que elabora el desarrollo?.

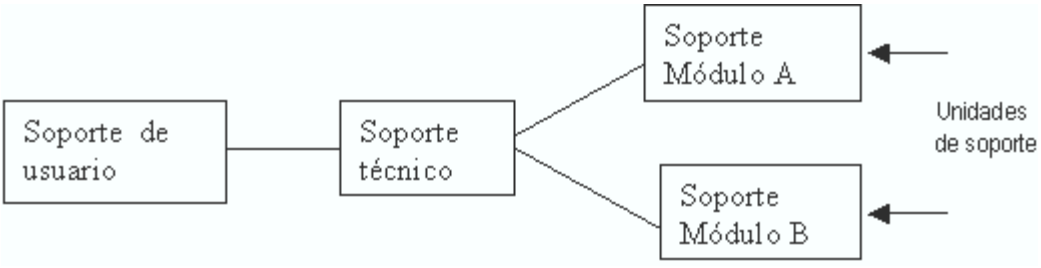
   La categoría predominante del mantenimiento es un conductor clave, si este es principalmente un mantenimiento correctivo, perfectivo o preventivo, entonces un equipo separado es quizás lo más apropiado; si se trata de un mantenimiento adaptativo, entonces sería preferible la disposición.

   b) Cómo hacer frente a esto con una carga de trabajo que está caracterizada por la fluctuación de requerimientos de recursos y solicitudes frecuentes para acciones de emergencia. Una respuesta sería que los niveles de personal fluctúen con la demanda. La solución ideal sería suavizar las fluctuaciones en primer lugar, con una planeación de las especificaciones del producto más detallada y con mejoras en las áreas de control de lista por prioridad y control de liberación.

5. **Reclutamiento y Motivación**: Mucha gente ve el mantenimiento como una actividad no glamorosa y esta actitud puede dificultar el reclutamiento y la motivación del personal hacia él mismo.

6. **Punto de vista del usuario**: Los encargados del mantenimiento y los usuarios finales de un sistema pueden tener puntos de vista diferentes acerca del desempeño del mismo. Un ejemplo es el tiempo requerido para corrección de un error, este es el lapso entre la recepción del informe del problema y la prueba exitosa de una versión modificada del sistema (para los encargados del mantenimiento). Es el tiempo entre la percepción inicial del problema y la instalación exitosa de la nueva versión (para los usuarios).

En la figura se muestra una red muy simple de funciones de mantenimiento. Es posible que una solicitud de usuario tenga que pasar a través de esas funciones antes de ser atendida. Cada función individual puede responder con rapidez, pero el punto de vista global de usuario puede ser pobre. El punto importante es saber qué es la red y estar en posición de administrar el mantenimiento desde el punto de vista del usuario.



Otro aspecto de la percepción del usuario es las respuestas a las solicitudes de adaptaciones relativamente pequeñas al sistema. Aunque no sea un cambio difícil, bien puede tener que esperar su turno en la lista de prioridad antes de que sea atendido y, de manera comprensible, el usuario puede sentirse agraviado como resultado.

La configuración del software

La configuración de un producto es el conjunto completo de elementos asociados con ese producto, incluyendo listas, código fuente, documentación de diseño, herramientas de construcción, casos de prueba, y así. El proceso de control de esos elementos para que estén disponibles cuando se requiera se conoce como administración de la configuración (CM). La CM se maneja como el requerimiento técnico más fundamental para un mantenimiento efectivo.

El grado de formalidad aplicada a la administración de la configuración variará con el tamaño y estado del proyecto al que se aplica. El nivel más bajo, estará todo en la cabeza de una sola empresa individual un desarrollo casual y rápido; el nivel más alto, se tratará de un elaborado esquema soportado por manuales formales y será un fuerte concerniente de la administración del proyecto.

Los componentes de la administración pueden separarse como sigue:

- Identificación de la configuración
- Control de cambio.
- Auditoria de la configuración.
- Contabilidad del estado de la configuración.

Identificación de la configuración: Es el proceso de asegurar que todos los elementos de la configuración sean descubiertos e identificados de manera única, y que la identificación incluya significados de distinción entre las diferentes versiones del mismo elemento. También proporciona el significado para registrar los cambios entre las versiones de un elemento.

En el periodo de planeación, la identificación de la configuración es responsable de cubrir las entregas que resultarán del proyecto; se utiliza para descubrir lo que pasará, así como registrar qué es lo que ha pasado.

Control de cambio: Cuando se hacen cambios en los elementos de la configuración, tiene que haber un conjunto de reglas que gobiernen la forma que son instigados, evaluados, aprobados e implementados. Esas reglas y procedimientos son conocidos de manera colectiva como el mecanismo de control de cambio.

Auditoria de la configuración: Una auditoria de la configuración es una verificación periódica para asegurar la correcta operación de la identificación de la configuración y las funciones de control de cambio, así como para asegurar que la configuración como existe refleje en forma adecuada los requerimientos y los planes.

Configuración del estado de la configuración: Se refiere a la colocación y presentación de la información de los otros tres componentes. Es la expresión de los

varios estados en la vida de un producto que puede ser referenciado por cualquiera que desee leer la historia del proyecto.

Mantenimiento en la operación

Las fases del mantenimiento de la operación son:

- Solicitudes de cambio.
- Evaluación de solicitudes.
- Diseño.
- Control de liberación.
- Construcción.
- Prueba.
- Distribución.

Solicitud de Cambio.

Da origen a tres elementos:

Existencia de la necesidad. Hay mucho interés de investigación en esta área, dirigido hacia la predicción de las características del programa referentes a cuantos errores pueden esperarse.

Percepción de la necesidad: En una situación ideal, esta fase transcurre muy rápido, pero si no es así los efectos en la satisfacción del usuario pueden ser significativos.

Notificación de la Necesidad: El interés aquí reside en la cantidad de detalle que puede ser comunicado y el mejor medio a utilizar para la comunicación.

Evaluación de la solicitud.

En esta fase, el beneficio del cambio propuesto se pone en la balanza contra el costo probable de la implementación, y la solicitud se pone en línea de espera de acuerdo con su prioridad resultante.

Los componentes de esta fase son:

- Aseguramiento del beneficio. La expresión de lo que se ganará con la implementación del cambio.
- Aseguramiento del costo. Cuánto se espera que cueste la implementación del cambio.
- Control de lista de prioridad. Las solicitudes prioritarias deben ser mantenidas en una manera conveniente a fin de que el esfuerzo subsecuente de mantenimiento pueda ser dirigido cuando es más costo-efectivo.

(Re) Diseño

Los componentes son muy parecido a los del desarrollo inicial del ciclo de vida, pero algunos adicionales son:

- *Evaluación de los requerimientos*
- *Reproducción de la falta*. Si la solicitud es para una corrección en el comportamiento del software, por lo general es necesario encontrar una forma de reproducción del problema.
- *Especificación.* El cambio se describe en términos del comportamiento.
- *Localización de la falta.* Para el mantenimiento correctivo la parte o partes ofensoras de código deben ser identificadas.
- *Diseño detallado.* Los cambios requeridos del código actual son diseñados y registrados.
- *Revisión de diseño.* Para asegurar que los requerimientos se satisfacen y que no se han generado nuevos errores.

Control de liberación.

Esta fase se refiere a crear una nueva versión del software, incorporando algunos o todos los cambios que estén disponibles.

Los componentes son:

- *Control de Tiempo de Liberación.* Este dirige la pregunta de en qué momento debe hacerse una nueva liberación de software.
- *Control de contenido de Liberación.* La decisión de cuáles de los cambios que están disponibles deben incluirse en le momento actual en una liberación particular.

Construcción

En esta fase, los cambios de código se aplican al código fuente y el resultado se compila para producir la nueva versión del software.

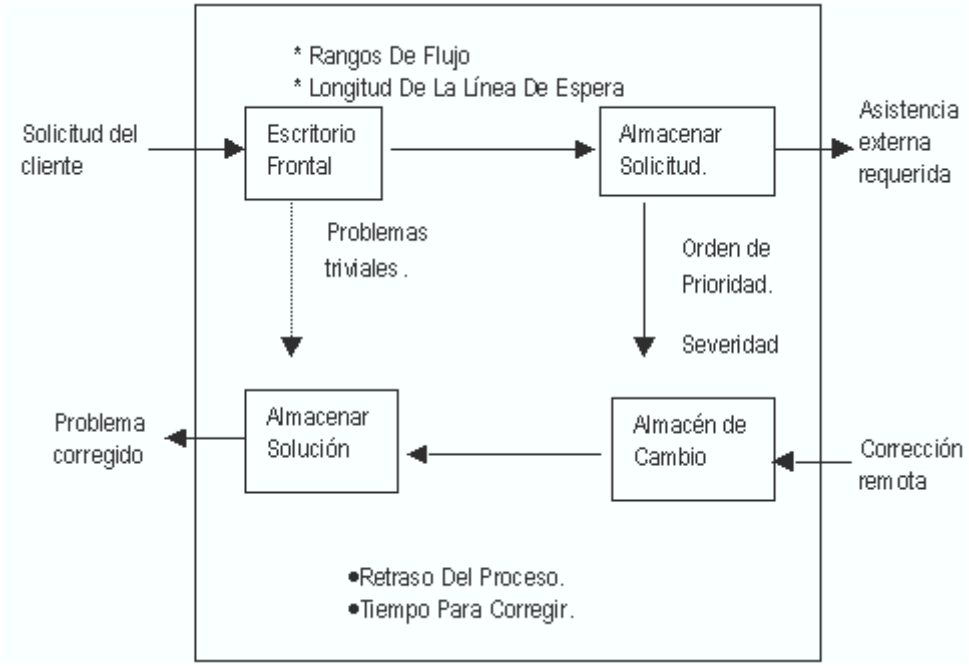
Es posible identificar al menos los siguientes componentes:

- *Preparación de la versión padre.* Los paso requeridos para asegurar que el código fuente y la versión que va ha ser modificada son, por lo tanto las versiones propuestas.
- *Edición y revisión.* Introducir los cambios y asegurar que no se han introducido errores en ese proceso.
- *Construcción del módulo.* Módulo significa cualquier subcomponente del sistema que pueda ser probado de manera independiente.
- *Integración:* Combinar módulos individuales para crear el sistema completo.
- *Archivar.* La preservación segura del código y la documentación una vez modificados.

Prueba

La prueba del cambio realizada en una parte del software cae en las siguientes tres categorías:

- *Probar la confidencialidad*. El efecto principal propuesto del cambio se demuestra.
- *Prueba de localización*. Se hace una revisión para ver que todos los efectos esperados de los cambios se observan en realidad.
- *Prueba de no localización*. La operación del sistema como un todo se verifica, en un intento de revelar cualquier efecto no deseado de los cambios.



Distribución

Los componentes de la distribución son:

- *Presentación*: El software se enviará como paquete, y la forma y contenido del paquete deben diseñarse así como para verificar su exactitud.
- *Réplica*: Las copias deben producirse para cada sitio donde se instala el software
- *Entrega*: En muchos casos esta es una tarea relativamente simple, pero los sistemas injertados de nuevo proporcionan un ejemplo contrario.
- *Instalación*: La nueva copia de software debe ser instalada y su operación verificada , y los registros del sitio deben mantenerse.

El modelo representado en la figura es útil en dos sentidos. En primer lugar, diferencia con claridad algunas de las actividades claves dentro del equipo de mantenimiento y proporciona una base para la asignación de roles de equipo .

El segundo uso de este modelo de equipo de mantenimiento es para clarificar los tipos de medidas que pueden aplicarse a fin de medir el proceso de mantenimiento.

Una cuestión final del mantenimiento en acción se refiere al uso de herramientas en el mantenimiento.

Mejoramiento del mantenimiento

Consideremos tres áreas significativas en las cuales hay una promesa significativa de qué problemas de mantenimiento puedan ocurrir con facilidad. Estas son:

- *Ingeniería Reverse*: Un medio de mejorar la práctica actual del software de mantenimiento.
- *Medición*: Un medio de establecer una comprensión más profunda del proceso de mantenimiento y determinación a través de datos objetivos de lo mejor que puede hacerse.
- *Diseño para el mantenimiento*: Pretender disminuir el mantenimiento futuro mediante la prevención de la etapa de diseño.

Ingeniería Reverse

La documentación del software debe producirse como un producto del proceso de desarrollo y manejarse más allá de un paquete completo, con el código fuente , para el equipo que mantendrá el programa.

El programador del mantenimiento tiene que reconstruir la documentación del programa a partir del código para adquirir una comprensión propia de cómo trabaja el sistema.

La “Ingeniería Reverse” es la habilidad para reconstruir la forma lógica del software a partir del código fuente. Las herramientas de soporte para la Ingeniería Reverse han comenzado a volverse disponibles a nivel comercial en los últimos años, sobre todo en le soporte de programas de COBOL. Hay un interés creciente en la Ingeniería REVERSE conforme esta se mueve con rapidez hacia una técnica establecida en algunas áreas, notablemente en los sistemas de procesamiento de datos.

Medición

Para llevar a cabo el mantenimiento de manera efectiva es esencial medir el proceso.

El objetivo de la medición es el establecimiento de relaciones entre la mantenibilidad de una parte del software y las prácticas de diseño que fueron usadas para producirlo.

Diseño para el Mantenimiento

La adopción de procedimientos de administración de calidad, tales como ISO 9000, reconoce la necesidad de procedimientos controlados de diseño y documentación.

A nivel técnico el ciclo está cerrado, el futuro de los puntos de mantenimiento de regreso a etapas previas del proceso de desarrollo y su correlación con uno de os requerimientos claves no-funcionales: la mantenibilidad.

Una lista de mantenimiento

La lista que se presenta en seguida está estructurada en cinco secciones principales:

El Sistema

- ¿Qué máquinas, sistemas operativos y lenguajes están involucrados? (Es esencial tener una imagen clara del equipo básico necesitado para recrear todas las instancias de un sistema liberado).
- ¿El sistema es de una sola vez o habrá varias instalaciones? (La réplica y entrega a varios sitios, a menudo con diferente documentación y requerimientos construidos, necesita ser planeada).
- ¿Cuál es la vida esperada del sistema? (Hay diferentes consideraciones, planes y prioridades en el mantenimiento de un sistema por un año como opuesto a 20 años).

El software

- ¿Qué tan grande es el programa? (Incluso un número aproximado de líneas de código es útil).
- ¿Cómo fue diseñado el sistema? (El método usado para crear el software es útil en la comprensión de su estructura y, de hecho, facilita su mantenimiento).
- ¿Algún componente del software es suplido por una tercera parte? (A parte de los lenguajes usados construir componentes puede implicar dependencia en la red de mantenimiento).

La organización del mantenimiento

- ¿Cómo y de qué forma se mantiene la documentación? (La documentación de diseño a menudo se utiliza con menor frecuencia en el mantenimiento que en los catálogos de referencia cruzada, etc.).
- ¿Dónde recaen las responsabilidades del mantenimiento? (Quién maneja las encuestas, primera línea y segunda línea de soporte).
- ¿Existe un esquema de informe de falla? (Debe haber un mecanismo para registrar y tratar con las solicitudes de cambio).
- ¿Existe algún estatuto de costo esperado del mantenimiento anual ? (Una comprensión de los costos reales de mantenimiento es vital en el ajuste de fase de la vida del producto).
- ¿Pueden recuperarse y construirse todas las versiones y variantes del sistema? (A menudo hay muchos productos similares, pero no idénticos, mantenidos en la misma área).
- ¿Hay productos definidos para CM? (Archivos de proyectos, librerías fuente, listas de compatibilidad, notas de liberación, etc. Todo necesita ser controlado. Debe haber procedimientos por cambiar).
- ¿Qué procedimientos de recuperación hay ?

Resumen

El mensaje principal ha sido que el mantenimiento no es un simple asunto de corrección de errores después que el software es liberado. Es un proceso continuo el mantener el software a tono con los requerimientos del usuario y el proceso de mantenimiento tiene que ser organizado a fin de enfrentar los cambios en la función, en el ambiente y lo probable, así como las fallas informadas.

Las implicaciones de esto son significativas.

Primero, es vital entender con claridad los tipos de mantenimiento requeridos y las fases por las que pasa un sistema de software.

Los tipos descritos aquí son:

1. Mantenimiento Correctivo (Mantenimiento de reparación).
2. Mantenimiento Adaptativo (Para hacer frente a un nuevo sistema operativo).
3. Mantenimiento Perfectivo (Mantenimiento para optimizar el desempeño, etc.).
4. Mantenimiento Preventivo (Para incrementar la robustez).

Las fases son:

1. Realce, en la cual la especificación del producto implica considerable alteración.
2. Madurez, en la que la especificación se mantiene relativamente estable y la actividad principal es el mantenimiento correctivo.
3. Obsolescencia, en la cual un producto continúa disponible y en uso, pero la habilidad para alterarlos se ha perdido.
4. Terminación, en la que el producto puede continuar en uso, pero todo control ha sido abandonado.

Segundo, es importante controlar el ciclo de modificación de un sistema en mantenimiento. Mucho del trabajo técnico llevado a cabo por el encargado del mantenimiento ha sido descubierto en el diseño y prueba.

HERRAMIENTAS CASE PARA EL DESARROLLO DE SOFTWARE

Hoy en día, muchas empresas se han extendido a la adquisición de herramientas CASE (Ingeniería Asistida por Computadora), con el fin de automatizar los aspectos clave de todo el proceso de desarrollo de un sistema, desde el principio hasta el final e incrementar su posición en el mercado competitivo, pero obteniendo algunas veces elevados costos en la adquisición de la herramienta y costos de entrenamiento de personal así como la falta de adaptación de la herramienta a la arquitectura de la información y a las metodologías de desarrollo utilizadas por la organización. Por otra parte, algunas herramientas CASE no ofrecen o evalúan soluciones potenciales para los problemas relacionados con sistemas o virtualmente no llevan a cabo ningún análisis de los requerimientos de la aplicación.

Sin embargo, CASE proporciona un conjunto de herramientas semiautomatizadas y automatizadas que están desarrollando una cultura de ingeniería nueva para muchas empresas. Uno de los objetivos más importante del CASE (a largo plazo) es conseguir la generación automática de programas desde una especificación a nivel de diseño.

Ahora bien, con la aparición de las redes de ordenadores en empresas y universidades ha surgido en el mundo de la informática la tecnología cliente / servidor. Son muchas de las organizaciones que ya cuentan con un número considerable de aplicaciones cliente / servidor en operación: Servidores de Bases de Datos y Manejadores de Objetos Distribuidos. Cliente / servidor es una tecnología de bajo costo que proporciona recursos compartidos, escalabilidad, integridad, encapsulamiento de servicios, etc. Pero al igual que toda tecnología, el desarrollo de aplicaciones cliente / servidor requiere que la persona tenga conocimientos, experiencia y habilidades en procesamiento de transacciones, diseño de base de datos, redes de ordenadores y diseño gráfica de interfase.

El objeto de estudio está centrado en determinar ¿cuáles son las influencias de las herramientas CASE en las empresas desarrolladoras de sistemas de información cliente / servidor? Y ¿cuáles son las tendencias actuales de las empresas fabricantes de sistemas cliente / servidor?.

Herramientas Case

De acuerdo con Kendall y Kendall la ingeniería de sistemas asistida por ordenador es la aplicación de tecnología informática a las actividades, las técnicas y las metodologías propias de desarrollo, su objetivo es acelerar el proceso para el que han sido diseñadas, en el caso de CASE para automatizar o apoyar una o mas fases del ciclo de vida del desarrollo de sistemas.

Cuando se hace la planificación de la base de datos, la primera etapa del ciclo de vida de las aplicaciones de bases de datos, también se puede escoger una herramienta CASE (Computer-Aided Software Engineering) que permita llevar a cabo el resto de tareas del modo más eficiente y efectivo posible. Una herramienta CASE suele incluir:

- Un diccionario de datos para almacenar información sobre los datos de la aplicación de bases de datos.
- Herramientas de diseño para dar apoyo al análisis de datos.
- Herramientas que permitan desarrollar el modelo de datos corporativo, así como los esquemas conceptual y lógico.
- Herramientas para desarrollar los prototipos de las aplicaciones.

El uso de las herramientas CASE puede mejorar la productividad en el desarrollo de una aplicación de bases de datos.

Historia

En la década de los setenta el proyecto ISDOS desarrolló un lenguaje llamado "Problem Statement Language" (PSL) para la descripción de los problemas de usuarios y las necesidades de solución de un sistema de información en un diccionario computarizado. Problem Statement Analyzer (PSA) era un producto asociado que analizaba la relación de problemas y necesidades.

Pero la primera herramienta CASE como hoy la conocemos fue “Excelerator” en 1984, era para PC. Actualmente la oferta de herramientas CASE es muy amplia y tenemos por ejemplo el EASYCASE o WINPROJECT. (Monografías.com)

Tecnología Case

La tecnología CASE supone la automatización del desarrollo del software, contribuyendo a mejorar la calidad y la productividad en el desarrollo de sistemas de información y se plantean los siguientes objetivos:

- Permitir la aplicación práctica de metodologías estructuradas, las cuales al ser realizadas con una herramienta se consigue agilizar el trabajo.
- Facilitar la realización de prototipos y el desarrollo conjunto de aplicaciones.
- Simplificar el mantenimiento de los programas.
- Mejorar y estandarizar la documentación.
- Aumentar la portabilidad de las aplicaciones.
- Facilitar la reutilización de componentes software.
- Permitir un desarrollo y un refinamiento visual de las aplicaciones, mediante la utilización de gráficos.

Automatizar:
El desarrollo del software
La documentación
La generación del código
El chequeo de errores
La gestión del proyecto

Permitir:
La reutilización del software
La portabilidad del software
La estandarización de la documentación

Componentes de una herramienta case

De una forma esquemática podemos decir que una herramienta CASE se compone de los siguientes elementos:

- Repositorio (diccionario) donde se almacenan los elementos definidos o creados por la herramienta, y cuya gestión se realiza mediante el apoyo de un Sistema de Gestión de Base de Datos (SGBD) o de un sistema de gestión de ficheros.
- Meta modelo (no siempre visible), que constituye el marco para la definición de las técnicas y metodologías soportadas por la herramienta.
- Carga o descarga de datos, son facilidades que permiten cargar el repertorio de la herramienta CASE con datos provenientes de otros sistemas, o bien generar a partir de la propia herramienta esquemas de base de datos, programas, etc. que pueden, a su vez, alimentar otros sistemas. Este elemento proporciona así un medio de comunicación con otras herramientas.
- Comprobación de errores, facilidades que permiten llevar a cabo un análisis de la exactitud, integridad y consistencia de los esquemas generados por la herramienta.
- Interfaz de usuario, que constará de editores de texto y herramientas de diseño gráfico que permitan, mediante la utilización de un sistema de ventanas, iconos y menús, con la ayuda del ratón, definir los diagramas, matrices, etc. que incluyen las distintas metodologías.

Estructura general de una herramienta case

La estructura CASE se basa en la siguiente terminología:

- CASE de alto nivel son aquellas herramientas que automatizan o apoyan las fases finales o superiores del ciclo de vida del desarrollo de sistemas como la planificación de sistemas, el análisis de sistemas y el diseño de sistemas.
- CASE de bajo nivel son aquellas herramientas que automatizan o apoyan las fases finales o inferiores del ciclo de vida como el diseño detallado de sistemas, la implantación de sistemas y el soporte de sistemas.
- CASE cruzado de ciclo de vida se aplica a aquellas herramientas que apoyan actividades que tienen lugar a lo largo de todo el ciclo de vida, se incluyen actividades como la gestión de proyectos y la estimación.

Estado Actual

En las últimas décadas se ha trabajado en el área de desarrollo de sistemas para encontrar técnicas que permitan incrementar la productividad y el control de calidad en cualquier proceso de elaboración de software, y hoy en día la tecnología CASE (Computer Aided Software Engineering) reemplaza al papel y al lápiz por el ordenador para transformar la actividad de desarrollar software en un proceso automatizado.

La tecnología CASE supone la –informatización de la informática—es decir –la automatización del desarrollo del software--, contribuyendo así a elevar la productividad y la calidad de en el desarrollo de los sistemas de información de forma análoga a lo que suponen las técnicas CAD/CAM en el área de fabricación.

En este nuevo enfoque que persigue mejorar la calidad del software e incrementar la productividad en el proceso de desarrollo del mismo, se plantean los siguientes objetivos:

1. Permitir la aplicación práctica de metodologías, lo que resulta muy difícil sin emplear herramientas.
2. Facilitar la realización de prototipos y el desarrollo conjunto de aplicaciones.
3. Simplificar el mantenimiento del software.

    - Mejorar y estandarizar la documentación.
    - Aumentar la portabilidad de las aplicaciones.
    - Facilitar la reutilización de componentes de software
    - Permitir un desarrollo y un refinamiento (visual) de las aplicaciones, mediante la utilización de controles gráficos (piezas de código reutilizables).

Integración de las herramientas case en el futuro

Las herramientas CASE evolucionan hacia tres tipos de integración:

1. La integración de datos permite disponer de herramientas CASE con diferentes estructuras de diccionarios locales para el intercambio de datos.
2. La integración de presentación confiere a todas las herramientas CASE el mismo aspecto.
3. La integración de herramientas permite disponer de herramientas CASE capaces de invocar a otras CASE de forma automática.

Clasificación de las herramientas case
No existe una única clasificación de herramientas CASE y, en ocasiones, es difícil incluirlas en una clase determinada. Podrían clasificarse atendiendo a:

- Las plataformas que soportan.
- Las fases del ciclo de vida del desarrollo de sistemas que cubren.
- La arquitectura de las aplicaciones que producen.
- Su funcionalidad.

CASE es una combinación de herramientas software (aplicaciones) y de metodologías de desarrollo :

1. Las herramientas permiten automatizar el proceso de desarrollo del software.
2. Las metodologías definen los procesos automatizar.

Una primera clasificación del CASE es considerando su amplitud :

TOOLKIT: es una colección de herramientas integradas que permiten automatizar un conjunto de tareas de algunas de las fases del ciclo de vida del sistema informático: Planificación estratégica, Análisis, Diseño, Generación de programas.

WORKBENCH: Son conjuntos integrados de herramientas que dan soporte a la automatización del proceso completo de desarrollo del sistema informático. Permiten cubrir el ciclo de vida completo. El producto final aportado por ellas es un sistema en código ejecutable y su documentación.

Una segunda clasificación es teniendo en cuenta las fases (y/o tareas) del ciclo de vida que automatizan:

UPPER CASE: Planificación estratégica, Requerimientos de Desarrollo Funcional de Planes Corporativos.
MIDDLE CASE: Análisis y Diseño.
LOWER CASE: Generación de código, test e implantación

Características Deseables De Una Case
Una herramienta CASE cliente / servidor provee modelo de datos, generación de código, registro del ciclo de vida de los proyectos, comunicación entre distintos ingenieros. Las principales herramientas son KnowledgeWare's Application Development Workbench, TI's, Information Engineering Facility (IEF), y Andersen Consulting's Foundation for Cooperative Processing.

Deberes de una herramienta CASE Cliente / servidor:

- Proporcionar topologías de aplicación flexibles. La herramienta debe proporcionar facilidades de construcción que permita separar la aplicación (en muchos puntos diferentes) entre el cliente, el servidor y más importante, entre servidores.
- Proporcionar aplicaciones portátiles. La herramienta debe generar código para Windows, OS/ 2, Macintosh, Unix y todas las plataformas de servidores conocidas. Debe ser capaz, a tiempo de corrida, desplegar la versión correcta del código en la máquina apropiada.
- Control de Versión. La herramienta debe reconocer las versiones de códigos que se ejecutan en los clientes y servidores, y asegurarse que sean consistentes. También, la herramienta debe ser capaz de controlar un gran número de tipos de objetos incluyendo texto, gráficos, mapas de bits, documentos complejos y objetos únicos, tales como definiciones de pantallas y de informes, archivos de objetos y datos de prueba y resultados. Debe mantener versiones de objetos con niveles arbitrarios de granularidad; por ejemplo, una única definición de datos o una agrupación de módulos.
- Crear código compilado en el servidor. La herramienta debe ser capaz de compilar automáticamente código 4GL en el servidor para obtener el máximo performance.
- Trabajar con una variedad de administradores de recurso. La herramienta debe adaptarse ella misma a los administradores de recurso que existen en varios servidores de la red; su interacción con los administradores de recurso debería ser negociable a tiempo de ejecución.

- Trabajar con una variedad de software intermedios. La herramienta debe adaptar sus comunicaciones cliente / servidor al software intermedio existente. Como mínimo la herramienta debería ajustar los temporizadores basándose en, si el tráfico se está moviendo en una LAN o WAN.
- Soporte multiusuarios. La herramienta debe permitir que varios diseñadores trabajen en una aplicación simultáneamente. Debe gestionarse los accesos concurrentes a la base de datos por diferentes usuarios, mediante el arbitrio y bloqueos de accesos a nivel de archivo o de registro.
- Seguridad. La herramienta debe proporcionar mecanismos para controlar el acceso y las modificaciones a los que contiene. La herramienta debe, al menos, mantener contraseñas y permisos de acceso en distintos niveles para cada usuario. También debe facilitar la realización automática de copias de seguridad y recuperaciones de las mismas, así como el almacenamiento de grupos de información determinados, por ejemplo, por proyecto o aplicaciones.
- Desarrollo en equipo, repositorio de librerías compartidas. Debe permitir que grupos de programadores trabajen en un proyecto común; debe proveer facilidades de check-in/ check-out registrar formas, widgets, controles, campos, objetos de negocio, DLL, etc.; debe proporcionar un mecanismo para compartir las librerías entre distintos realizadores y múltiples herramientas; Gestiona y controla el acceso multiusuario a los datos y bloquea los objetos para evitar que se pierdan modificaciones inadvertidamente cuando se realizan simultáneamente.

Factores asociados a la implantación de las herramientas case

La difusión de las innovaciones en esta área ha comenzado a estudiarse a partir de los años 1940. Por ello, existen estudios teóricos al respecto, realizándose evaluaciones, adopción e implementación tecnológica.

Existe un amplio cuerpo de investigaciones disponibles sobre la adopción de innovaciones. Muchos de los estudios sobre innovación se han analizado bajo dos perspectivas: adopción y difusión (Kimberly, 1981). Mientras unos estudios usan la perspectiva de la adopción para evaluar la receptividad y los cambios de la organización o sociedad por la innovación, otros usan la perspectiva de la difusión para intentar entender por qué y cómo se difunde y qué características generales o principales de la innovación son aceptadas.

Conclusión

Sin lugar a dudas las herramientas CASE han venido a revolucionar la forma de automatizar los aspectos clave en el desarrollo de los sistemas de información, debido a la gran plataforma de seguridad que ofrecen a los sistemas que las usan y es que éstas, brindan toda una gama de componentes que incluyen todas o la mayoría de los requisitos necesarios para el desarrollo de los sistemas, han sido creadas con una gran exactitud en torno a las necesidades de los desarrolladores de sistemas para la automatización de procesos incluyendo el análisis, diseño e implantación.

Las Herramientas CASE se clasifican por su amplitud en: TOOLKIT, WORKBENCH además también se pueden dividir teniendo en cuenta las fases del ciclo de vida que automatizan: UPPER CASE, MIDDLE CASE, LOWER CASE.

Debido a la gran demanda que tienen las CASE su exigencia en cuanto a su uso ha ido aumentando, por lo que toda CASE debe entre otras cosas:

- Proporcionar topologías de aplicación flexibles
- Proporcionar aplicaciones portátiles
- Brindar un Control de versión
- Crear código compilado en el servidor
- Dar un Soporte multiusuario
- Ofrecer Seguridad

Desde que se crearon éstas herramientas (1984) hasta la actualidad, las CASE cuentan con una credibilidad y exactitud que tienen un reconocimiento universal, siendo usadas por cualquier desarrollador y / o programador que busca un resultado óptimo y eficiente, pero sobre todo que busca esa minuciosidad necesaria de los procesos y entre los procesos.

BIBLIOGRAFÍA

ALFREDO WEITZENFELD (2004) Ingeniería de software orientada a objetos con UML JAVA, e Internet. Editorial Thomson. México.

IAN SOMMERVILLE. (1998). Ingeniería de Software. Addison Wesley Iberoamericana, Wilnington (EE.UU.)

PRESSMAN, ROGER S. (2002). Ingeniería del software, un enfoque práctico. Madrid: McGraw-Hill. 601p. 5ta edición