UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN CIENCIAS Y SISTEMAS

METODOLOGÍAS PARA EL DESARROLLO DE *SOFTWARE*: UN ANÁLISIS COMPARATIVO

PEDRO PABLO HERNÁNDEZ RAMIREZ

ASESORADO POR: INGA. VIRGINIA VICTORIA TALA AYERDI

GUATEMALA, OCTUBRE DE 2004

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

METODOLOGÍAS PARA EL DESARROLLO DE *SOFTWARE*: UN ANÁLISIS COMPARATIVO

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA

POR

PEDRO PABLO HERNÁNDEZ RAMIREZ

ASESORADO POR: INGA. VIRGINIA VICTORIA TALA AYERDI

AL CONFERÍRSELE EL TÍTULO DE
INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, OCTUBRE DE 2004

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Sydney Alexander Samuels Milson
VOCAL I	Ing. Murphy Olympo Paiz Recinos
VOCAL II	Ing. Amahám Sánchez Álvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Ing. Marlon Antonio Pérez Turk
EXAMINADOR	Ing. Édgar René Ornelyz Oil
EXAMINADOR	Ing. Luis Alberto Vettorazzi España
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado

METODOLOGÍAS PARA EL DESARROLLO DE *SOFTWARE*:
UN ANÁLISIS COMPARATIVO

Tema que me fuera asignado por la dirección de la carrera de Ingeniería en Ciencias y Sistemas en febrero de 2003.

Pedro Pablo Hernández Ramirez

DEDICATORIA

A Dios

Por darme la vida, las oportunidades y sobre todo por llevarme por el sendero del saber, hasta llegar a una de mis metas y ser mi fiel guía en todo momento de mi vida.

A mis padres

Juan José Hernández Rojas y Efraína Ramírez Ortiz, por brindarme su apoyo incondicional en todo momento porque esta meta alcanzada es la bendición de sus esfuerzos.

A mis hermanos

Amanda y Juan José, con especial afecto, Lourdes Azucena Celeste, Filiberto (Q.E.P.D.), con cariño.

A mi asesora

Inga. Virginia Victoria Tala Ayerdi, por compartir su experiencia, asesorar este trabajo, brindarme sus consejos que me han servido tanto en mi vida universitaria como profesional.

A mis amigos

Por su amistad, compañerismo y tantos momentos compartidos. Porque cada uno de ellos alcance sus metas.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

GLOSARIO

Abstracción Se refiere a quitar las propiedades y acciones de un objeto para dejar sólo aquellas que sean necesarias.

Acumulador Campo o variable que sirve para llevar una suma o cuenta de diferentes valores.

Agente Pequeño programa inteligente que facilita la operatoria del usuario. Un ejemplo de agente son los asistentes que existen en la mayoría de los programas modernos.

Algoritmo Conjunto de sentencias o instrucciones en lenguaje nativo, los cuales expresan la lógica de un programa.

Algoritmo cualitativo Son aquellos que al resolver un problema no ejecutan operaciones matemáticas en el desarrollo de algoritmo.

Algoritmo cuantitativo Son aquellos algoritmos que ejecutan operaciones numéricas durante su ejecución.

API *Aplication Program Interface.* Es el conjunto de rutinas del sistema que se pueden usar en un programa para la gestión de entrada y salida, gestión de ficheros etc.

Aplicación Programa que realiza una serie de funciones y con el cual trabajamos en el ordenador.

Applet Pequeño programa hecho en lenguaje Java.

Archivo Unidad de información almacenada en el disco con un nombre específico. Tienen una extensión consistente en tres caracteres que lo identifican en su tipo o lo relacionan con un programa determinado.

ASCII *American Standard Code for Information Interchange* o Código Americano Normado para el Intercambio de Información. Es un conjunto de caracteres, letras y símbolos que permite una base común de comunicación. Incluye las letras normales del alfabeto, pero excluye la ñ y toda letra acentuada. Cada símbolo posee un número asignado que es común en todos los países.

Backup Aplicación de copia de seguridad de ficheros, carpetas o unidades completas que permite dividir la información o ficheros en varios disquetes y que además la comprime.

Base de datos Es un almacenamiento colectivo de las bibliotecas de datos que son requeridas y organizadas para cubrir sus requisitos de procesos y recuperación de información.

BASIC — *Beginners All Purpus Symbolic Instruction Code*. Lenguaje de instrucciones simbólicas de propósito general para principiantes, está disponible en modo compilador e intérprete, siendo este último el más popular para el usuario circunstancial y para el programador principiante.

Beta — Versión anterior a la Alfa y que puede ser la versión definitiva que se comercializará en un determinado tiempo.

Binario — Código básico de la informática que reduce todo tipo de información a cadenas de ceros y unos, que rigen las instrucciones y respuestas del microprocesador.

BIOS — *Basic Input Output System*. Sistema básico de entrada y salida. Programa residente normalmente en *Eprom* que controla las interacciones básicas entre el *hardware* y el *software*.

Bit — Un dígito simple de un número binario (1 ó 0) en el computador.

Browser — Programa que permite leer documentos en la *Web* y seguir enlaces (*links*) de documento en documento de hipertexto. Los navegadores hacen pedidos de archivos (páginas y otros) a los *Servers* de *Web* según la elección del usuario y luego muestran en el monitor el resultado del pedido en forma multimedial. Entre

	los más conocidos se encuentran el *Netscape Navigator, Microsoft Explorer y Mosaic.*
Buffer	Memoria intermedia, una porción reservada de la memoria, que se utiliza para almacenar datos mientras son procesados.
Byte	Grupo de bits adyacentes operados como una unidad, (grupos de 8 bits).
Caché	Almacenamiento intermedio o temporal de información. Por ejemplo, un navegador posee un caché donde almacena las últimas páginas visitadas por el usuario y, si alguna se solicita nuevamente, el navegador mostrará la que tiene acumulada en lugar de volver a buscarla en Internet. El término se utiliza para denominar todo depósito intermedio de datos solicitados con mayor frecuencia.
CAD	*Computer Aided Design*. Diseño asistido por ordenador.
Campo	Es el espacio en la memoria que sirve para almacenar temporalmente un dato durante el proceso. Su contenido varía durante la ejecución del programa.
Campo alfanumérico	El que puede almacenar cualquier carácter: dígito, letra, o símbolo especial.

Campo numérico El que sólo puede almacenar valores dígitos.

Caso de uso Es un diagrama que representa una estructura que sirve para describir la forma en que un sistema lucirá para los usuarios.

CGI *Common Gateway Interface* o interface de acceso común. Programas usados para hacer llamadas a rutinas o controlar otros programas o bases de datos desde una página *Web*.

Client side* CGI *script *Script* CGI que se ejecuta o corre en el cliente.

Cliente Computadora o programa que se conecta a servidores para obtener información. Un cliente sólo obtiene datos, no puede ofrecerlos a otros clientes sin depositarlos en un servidor. La mayoría de las computadoras que las personas utilizan para conectarse y navegar por Internet son clientes.

Cliente / Servidor Sistema de organización de interconexión de computadoras según el cual funciona Internet, así como otros tantos sistemas de redes. Se basa en la separación de las computadoras miembros en dos categorías: las que actúan como servidores (oferentes de información) y otras que actúan como clientes (receptores de información).

Código de máquina Para que se pueda ejecutar un programa, debe estar en lenguaje de máquina de la computadora que lo está ejecutando.

Código fuente Programa en su forma original, tal y como fue escrito por el programador, el código fuente no es ejecutable directamente por el computador, debe convertirse en lenguaje de máquina mediante compiladores, ensambladores o intérpretes.

Coma flotante Cálculo que realiza el procesador de operaciones con decimales.

Compilador Programa de computadora que produce un programa en lenguaje de máquina, de un programa fuente que generalmente esta escrito por el programador en un lenguaje de alto nivel.

Componente Representa una parte de nuestro sistema, por ejemplo, una base de datos, archivos, librerías, etcétera, que tiene una función específica.

Constante Valor o conjunto de caracteres que permanecen invariables durante la ejecución del programa.

Cracker Persona que se especializa en violar medidas de seguridad de una computadora o red de computadoras, venciendo claves de acceso y defensas para obtener información que cree valiosa. El *cracker* es

considerado un personaje ruin y sin honor, a diferencia del *hacker*.

Cross-platform Programa o dispositivo que puede utilizarse sin inconvenientes en distintas plataformas de *hardware* y sistemas operativos. Un programa en lenguaje Java posee esta característica.

Dato El término que usamos para describir las señales con las cuales trabaja la computadora es dato. Aunque las palabras dato e información muchas veces son usadas indistintamente, existe una diferencia importante entre ellas. En un sentido estricto, los datos son las señales individuales en bruto y sin ningún significado que manipulan las computadoras para producir información.

Default Opción que un programa asume si no se especifica lo contrario. También llamado valores predeterminados.

Depurador Es un programa que asiste en la depuración de un programa, sirve para la búsqueda y detección de errores.

Diagrama de flujo Es la representación gráfica de una secuencia de instrucciones de un programa que ejecuta un computador para obtener un resultado determinado.

Driver	Significa "controlador". Es el *software* adicional necesario para controlar la comunicación entre el sistema y un cierto dispositivo físico, tal como un monitor o una impresora.
Editor	Es un *software* empleado para crear y manipular archivos de texto, tales como programas en lenguaje fuente, lista de nombres y direcciones.
Encapsulamiento	Se refiere a que un objeto puede ocultar su funcionalidad a otros objetos.
Encriptación	Método para convertir los caracteres de un texto de modo que no sea posible entenderlo si no se lo lee con la clave correspondiente. Es utilizado para proteger la integridad de información secreta en caso de que sea interceptada. Uno de los métodos más conocidos y seguros de encriptación es el PGP.
Entorno gráfico	Sistema operativo en el que la información que aparece en pantalla aparece representada en forma gráfica, como es el caso de *Windows*.
Escalabilidad	Capacidad de crecimiento de la computadora.
Escáner	Dispositivo periférico que copia información impresa mediante un sistema óptico de lectura. Permite convertir imágenes, por ejemplo de fotografías, en imágenes tratables y que pueden ser almacenadas por

la computadora. El proceso de conversión se denomina digitalización. El término inglés *scanner* significa explorar o rastrear.

Firewall Conjunto de programas de protección y dispositivos especiales que ponen barreras al acceso exterior a una determinada red privada. Es utilizado para proteger los recursos de una organización de consultas externas no autorizadas.

Firmware Son pequeños programas que por lo general vienen en un chip en el *hardware*, como es el caso de la ROM BIOS.

Formateo Proceso por el que se adapta la superficie magnética de un disco para aceptar la información bajo un sistema operativo determinado. En el proceso de formateado se colocan las marcas lógicas que permitirán localizar la información en el disco y las marcas de sincronismo, además de comprobar la superficie del disco.

Freeware Política de distribución gratuita de programas, utilizada para gran parte del *software* de Internet. En general, estos programas son creados por un estudiante o alguna organización (usualmente una universidad) con el único objetivo de que mucha gente en el mundo pueda disfrutarlos. No son necesariamente sencillos: muchos de ellos son complejos y han llevado cientos

de horas de desarrollo. Ejemplos de *freeware* son el sistema operativo Linux o el PGP (*Pretty Good Privacy*, un *software* de encriptación), que se distribuyen de este modo.

Función — En programación, una rutina que hace una tarea particular. Cuando el programa pasa el control a una función, ésta realiza la tarea y devuelve el control a la instrucción siguiente a la que llamó.

Gigabyte — Medida de 1.000 MB (unos 1.000 millones de caracteres aproximadamente).

GUI — *Graphic User Interface*. Interface gráfico de usuario.

Gurú — Persona con muchos conocimientos sobre un tema, en general, técnico.

Hacker — Experto técnico en algún tema relacionado con comunicaciones o seguridad; de alguna manera, es también un gurú. Los *hackers* suelen dedicarse a violar claves de acceso por pura diversión, o para demostrar fallas en los sistemas de protección de una red de computadoras, casi como un deporte. Los *hackers* son muy respetados por la comunidad técnica de Internet, a diferencia de los *crackers*.

Hardware	Componente físico de la computadora. Por ejemplo el monitor, la impresora o el disco rígido. El *hardware* por sí mismo no hace que una máquina funcione. Es necesario, además, instalar un *software* adecuado.
Información	Es lo que se obtiene del procesamiento de datos, es el resultado final.
Instrucción o sentencia	Conjunto de caracteres que se utilizan para dirigir un sistema de procesamiento de datos en la ejecución de una operación.
Inteligencia artificial	Rama de la computación que analiza a la computadora y sus posibilidades de poseer inteligencia. La IA estudia las habilidades inteligentes de razonamiento, capacidad de extracción de conclusiones y reacciones ante nuevas situaciones de las computadoras y sus programas. El razonamiento es parecido al del cerebro humano (no es lineal, se aprende de cada situación).
Interface	Cara visible de los programas. La interface abarca las pantallas y su diseño, el lenguaje usado, los botones y los mensajes de error, entre otros aspectos de la comunicación computador - persona.
Internet	La red de computadoras más extendida del planeta, que conecta y comunica a más de 50 millones de personas. Nació a fines de los años sesenta como ARPANet y se convirtió en un revolucionario medio

de comunicación. Su estructura técnica se basa en millones de computadoras que ofrecen todo tipo de información. Estas computadoras, encendidas las 24 horas, se llaman servidores y están interconectadas entre sí en todo el mundo a través de diferentes mecanismos de líneas dedicadas. Sin importar qué tipo de computadoras son, para intercomunicarse utilizan el protocolo TCP/IP. Las computadoras que utilizan las personas para conectarse y consultar los datos de los servidores se llaman clientes, y acceden en general a través en un tipo de conexión llamado dial-in, utilizando un módem y una línea telefónica.

InterNIC — *Internet Network Information Center* o Centro de información de red de Internet. Centro de información que almacena documentos de Internet: RFCs y borradores de documentos. Organismo que se ocupa de otorgar grupos de números IP y direcciones electrónicas a cada organización que desee conectarse a Internet, garantizando que sean únicas. Más datos en http://www.internic.net/.

Intérprete — Dispositivo o programa que recibe una por una las sentencias de un programa fuente, la analiza y la convierte en lenguaje de máquina si no hay errores en ella. También se puede producir el listado de las instrucciones del programa.

Intranet Utilización de la tecnología de Internet dentro de la red local (LAN) y / o red de área amplia (WAN) de una organización. Permite crear un sitio público donde se centraliza el acceso a la información de la compañía. Bien utilizada, una intranet permite optimizar el acceso a los recursos de una organización, organizar los datos existentes en las computadoras de cada individuo y extender la tarea colaborativa entre los miembros de equipos de trabajo. Cuando una intranet extiende sus fronteras más allá de los límites de la organización, para permitir la intercomunicación con los sistemas de otras compañías, se la llama extranet.

IP Protocolo de Internet definido en el RFC 791. Confirma la base del estándar de comunicaciones de Internet. El IP provee un método para fragmentar y rutear la información. Es inseguro, ya que no verifica que todos los fragmentos del mensaje lleguen a su destino sin perderse en el camino. Por eso, se complementa con el TCP.

IP número o dirección IP Address. Dirección numérica asignada a un dispositivo de hardware conectado a Internet, bajo el protocolo IP. La dirección se compone de cuatro números, y cada uno de ellos puede ser de 0 a 255, las direcciones IP se agrupan en clases.

ISA *Industry Standard Arquitecture* o arquitectura estándar de la industria. Es la arquitectura adoptada en las PC compatibles y que define cómo deben ser las tarjetas que pueden conectarse a ellas.

ISO *International Standard Organization*. Organización internacional de estándares.

Java Lenguaje de programación creado por *Sun Microsystems*. Desde su aparición, Java se perfila como un probable revolucionario de la red. Como lenguaje es simple, orientado a objetos, distribuido, interpretado, robusto, seguro, neutral con respecto a la arquitectura, portable, *multithreaded* y dinámico. Java es un lenguaje de programación, un *subset* seguro de C++. Subset, porque algunas instrucciones (como las que tienen que ver con la administración de memoria) no se pueden usar. Seguro, porque agrega características de seguridad a los programas. Un *applet* de Java se baja automáticamente con la página *Web* y es compilado y ejecutado en la máquina local. Permite, entre otras cosas, agregar animación e interactividad a una página *Web*, pero su característica más importante es que un programa escrito en Java puede correr en cualquier computadora. Para más datos http://java.sun.com/.

Javascript	Lenguaje de *scripts* para utilizar en páginas *Web* desarrollado por *Netscape*. Permite aumentar la interactividad y la personalización de un sitio.
Kernel	Núcleo básico del sistema operativo, a partir del cual se establecen las distintas capas para su integración con el *hardware*, para la entrada y salida de datos, etc.
***Kilobyte* (Kb)**	Medida que equivale a 1.024 *bytes*.
LAN	*Local Area Network* o red de área local. Red de computadoras interconectadas, distribuida en la superficie de una sola oficina o edificio. También llamadas redes privadas de datos. Su principal característica es la velocidad de conexión.
LBA	Modo especial de direccionamiento del disco duro con el que se puede acceder a particiones de más de 528 *Mbytes.*
Linux	Versión *freeware* del conocido sistema operativo Unix. Es un sistema multitarea multiusuario de 32 bits para computadores personales.
Log	Archivo que registra movimientos y actividades de un determinado programa (*logfile*). Utilizado como mecanismo de control y estadística.

Lógica	Es una secuencia de operaciones realizadas por el hardware o por el *software*.
Lógica del *hardware*	Son los circuitos y chips que realizan las operaciones de control de la computadora.
Lógica del *software*	Lógica del programa, es la secuencia de instrucciones en un programa.
Login	Proceso de seguridad que exige que un usuario se identifique con un nombre (*User-ID*) y una clave, para poder acceder a una computadora o a un recurso.
LU	*Logic Unit*. Unidad Lógica.
Mainframe	Nombre con que se designan a las grandes computadoras que funcionan en sistemas centralizados.
MAN	*Metropolitan Area Network* o red de área metropolitana. Red que resulta de varias redes locales interconectadas por un enlace de mayor velocidad o *backbone*. Una MAN ocupa un área geográfica más extensa que una LAN, pero más limitada que una WAN.
***Megabyte* (Mb)**	Medida que equivale a 1.024 Kb, aproximadamente un millón de caracteres.

Memoria RAM — *Random Access Memory* o memoria de acceso aleatorio cuyo contenido permanecerá presente mientras el computador permanezca encendido.

Memoria ROM — *Read Only Memory* o memoria de sólo lectura. Chip de memoria que solo almacena permanentemente instrucciones y datos de los fabricantes.

Menú — Lista de comandos que aparece en la parte superior de las ventanas representadas por un nombre con una letra subrayada, y que sirve para dar instrucciones a los programas o para comunicarnos con ellos por medio de éstos.

Menú contextual — Lista de comandos que aparece al hacer clic con el botón derecho del ratón sobre un objeto.

Norma — Conjunto de reglas sobre algún producto o servicio que garantiza uniformidad en todo el mundo, en cualquier sistema en el que se implemente. Existen dos tipos de normas: la estándar (o normada), generada por comités especiales, y la de facto (o impuesta), que se acepta cuando un producto, debido a su uso, se convierte en universal. Los tres organismos más activos en el desarrollo de normas son: la ISO (*International Standards Organization*), la IEE (*American Institution of Electrical and Electronic Engineers*) y la CCITT (*International Telegraph and Telephone Consultative Comitee*). Las normas son la

base de los sistemas abiertos.

Paquete — Elemento que sirve para organizar los distintos elementos en un diagrama.

Password — Palabra utilizada para validar el acceso de un usuario a una computadora servidor.

Periféricos — Cualquier dispositivo de hardware conectado a una computadora.

PGP — *Pretty Good Privacy*. *Software* de encriptación *freeware* muy utilizado, desarrollado por Paul Zimmerman. Se basa en un método de clave pública y clave privada, y es óptimo en cuanto a seguridad. Su eficacia es tal que los servicios de inteligencia de varios países ya lo han prohibido. Más datos en http://www.pgp.com/.

Polimorfismo — Se da cuando existe un método con el mismo nombre en distintas clases, y cada uno realiza acciones diferentes.

Programa — Sinónimo de *software*. Conjunto de instrucciones que se ejecutan en la memoria de una computadora para lograr algún objetivo. Creados por equipos de personas en lenguajes especiales de programación, y se les diseña una interface de usuario para que puedan interactuar con las personas que los utilicen.

Programa ejecutable	Los archivos de programa a menudo se denominan programas ejecutables, puesto que, al teclear su nombre o al hacer clic sobre el icono que le corresponda en un entorno gráfico, logra que la computadora cargue y corra, o ejecute las instrucciones del archivo.
Programa ensamblador	Es un programa de computador preparado por un programador que toma las instrucciones que no estén en lenguaje de máquina y las convierte en una forma que puede ser usada por el computador.
Programador	Un individuo que diseña la lógica y escribe las líneas de código de un programa de computadora.
Programador de aplicaciones	Persona que escribe programas de aplicación en una organización. La mayoría de los programadores son programadores de aplicación.
Programador de sistemas	En el departamento de procesamiento de datos de una gran organización, técnico experto en parte o en la totalidad del *software* de sistemas de computadora, tal como el sistema operativo, el programa de control de red y el sistema de administración de base de datos. Los programadores de sistemas son responsables del rendimiento eficiente de los sistemas de computación.

Protocolo Conjunto de reglas formuladas para controlar el intercambio de datos entre dos entidades comunicadas. Pueden ser normados (definidos por un organismo capacitado, como la CCITT o la ISO) o de facto (creados por una compañía y adoptados por el resto del mercado).

Pseudo código Herramienta de análisis de programación. Versiones falsificadas y abreviadas de las actuales instrucciones de computador que son escritas en lenguaje ordinario natural.

Red Dos o más computadoras conectadas para cumplir una función, como compartir periféricos (impresoras), información (datos, sistema de ventas) o para comunicarse (correo electrónico). Existen varios tipos de redes: según su estructura jerárquica se catalogan en redes cliente / servidor, con computadoras que ofrecen información y otras que sólo consultan información, y las *peer-to-peer*, donde todas las computadoras ofrecen y consultan información simultáneamente. A su vez, según el área geográfica que cubran, las redes se organizan en LANs , MANs ó WANs.

Registro Es un grupo de campos relacionados que se usan para almacenar datos acerca de un tema (registro maestro) o actividad (registro de transacción).

R-login	*Remote Login* o acceso remoto. Acceso a un *Server* desde un sistema remoto.
Rutina	Es el conjunto de instrucciones dentro del mismo programa, que se puede llamar a ejecución desde diferentes partes del mismo programa.
Script	Programa no compilado realizado en un lenguaje de programación sencillo.
Server	Computadora que pone sus recursos (datos, impresoras, accesos) al servicio de otras a través de una red.
Server side* CGI *script	*Script* CGI que se ejecuta o corre en el servidor.
SET	*Secure Electronic Transactions* o Transacciones electrónicas seguras. Un estándar para pagos electrónicos encriptados que está siendo desarrollado por *Mastercard*, Visa y otras empresas. Similar al SSL.
Shareware	Política de distribución de programas donde se tiene derecho a probar un *software* por un determinado período antes de decidir comprarlo. El importe a abonar por el programa es en general bajo, prácticamente nominal.

Sistema operativo Conjunto de programas que se encarga de coordinar el funcionamiento de una computadora, cumpliendo la función de interface entre los programas de aplicación, circuitos y dispositivos de una computadora.

Sistemas abiertos Conjunto de computadoras de distintas marcas interconectadas, que utilizan el mismo protocolo normado de comunicación. El protocolo estándar más difundido es el TCP/IP.

Software Componentes intangibles (programas) de las computadoras. Complemento del *hardware*. El *software* más importante de una computadora es el sistema operativo.

SSL *Secure Socket Layer* o capa de seguridad. Estándar para transacciones electrónicas encriptadas que está siendo ampliamente utilizado para hacer negocios vía la red.

Subrutina Programa (conjunto de instrucciones) que desde otro programa se pueden llamar a ejecución o bien grupo de instrucciones que realizan una función especifica, tal como una función o marco. Una subrutina grande se denomina usualmente módulo o procedimiento, pero todos los términos se utilizan de manera alternativa.

TCP — *Transmission Control Protocol* o protocolo de control de transmisión. Conjunto de protocolos de comunicación que se encargan de la seguridad y la integridad de los paquetes de datos que viajan por *Internet*. Complemento del IP en el TCP/IP.

TCP/IP — *Transmission Control Protocol / Internet Protocol* o protocolo de control de transmisión / protocolo *Internet*. Usados para organizar computadoras en redes. Norma de comunicación en *Internet*, compuesta por dos partes: el TCP/IP. El IP desarma los envíos en paquetes y los rutea, mientras que el TCP se encarga de la seguridad de la conexión, comprueba que los datos lleguen todos, completos, y que compongan finalmente el envío original.

Telnet — Programa que permite el acceso remoto a un *host*. Utilizado para conectarse y controlar computadoras ubicadas en cualquier parte del planeta.

Terabyte — Unidad de almacenamiento futura, equivalente a más de un trillón de *bytes*.

UML — El *Unified Modeling Language* o lenguaje unificado de modelado. Es una metodología orientada a objetos y fue desarrollada por Grany Booch, James Rumbaugh e Ivar Jacobson, en los años ochenta y principios de los noventa.

Unidad Dispositivo físico de almacenamiento de los datos.

Upgrade Actualización de un programa.

Usuario Cualquier persona que interactúa con la computadora a nivel de aplicación. Los programadores, operadores y otro personal técnico no son considerados usuarios cuando trabajan con la computadora a nivel profesional.

Variable En programación es una estructura que contiene datos y recibe un nombre único dado por el programador, mantiene los datos asignados a ella hasta que un nuevo valor se le asigne o hasta que el programa termine.

Virus Pequeños programas de computadora que tienen la capacidad de auto duplicarse y residir en otros programas. Una vez que se difunden, los virus se activan bajo determinadas circunstancias y, en general, provocan algún daño o molestia.

W3C *World Wide Web Consortium*. Organización que desarrolla estándares para guiar la expansión de la *Web*. Su Website es http://www.w3.org/.

WAN *Wide Area Network* o red de área amplia. Resultante de la interconexión de varias redes locales localizadas en diferentes ciudades o países, comunicadas a través de conexiones públicas.

WWW *World Wide Web* o W3. Conjunto de servidores que proveen información organizada en *sites*, cada uno con cierta cantidad de páginas relacionadas. La *Web* es una forma novedosa de organizar toda la información existente en *Internet* a través de un mecanismo de acceso común de fácil uso, con la ayuda del hipertexto y la multimedia. El hipertexto permite una gran flexibilidad en la organización de la información, al vincular textos disponibles en todo el mundo. La multimedia aporta color, sonido y movimiento a esta experiencia. El contenido de la *Web* se escribe en lenguaje HTML y puede utilizarse con intuitiva facilidad mediante un programa llamado navegador. Se convirtió en el servicio más popular de la red y se emplea cotidianamente para los usos más diversos: desde leer un diario de otro continente hasta participar de un juego grupal.

Worm Tipo de programa similar al virus que se distribuye en una red. Su objetivo es generalmente afectar o dañar el funcionamiento de las computadoras.

RESUMEN

Una aplicación de *software*, es un programa que fue desarrollado con el objetivo de resolver un problema específico, una necesidad. Por ejemplo, llevar el control de compras y ventas de un almacén. En este caso, los posibles clientes y proveedores son limitados y el volumen de información manejado es pequeño o mediano.

Sin embargo, debido a la rapidez con que la tecnología avanza, este sistema se vuelve obsoleto al no estar diseñado para manejar las mismas u otras necesidades, en otras condiciones. Por ejemplo, la *Internet*, si este mismo almacén decide incursionar en el comercio electrónico, en donde los clientes potenciales no se pueden cuantificar debido al gran número de usuarios y su constante crecimiento.

Una posible solución sería el adaptar el sistema ya existente a las nuevas necesidades. Pero si el sistema no está diseñado para funcionar en los protocolos que gobiernan la *Internet*, el sistema debe de ser desarrollado por completo.

Esto es lo que sucede con las aplicaciones que son desarrolladas sin tomar en cuenta todos los factores, tanto internos como externos, en un proyecto o el utilizar una metodología de desarrollo inadecuada.

En este trabajo se explican las diversas metodologías de desarrollo de *software*, tanto tradicionales como evolutivas; sus respectivas etapas de desarrollo y características generales.

Finalmente, se hace una comparación entre todas las metodologías y los factores que están presentes en un proyecto. El único objetivo de recomendar una metodología

de desarrollo que mejor se adapte a sus requerimientos o necesidades, para hacer un sistema que se pueda adaptar a los cambios que surgen con el tiempo.

OBJETIVOS

Generales

- Proporcionar un documento que compare las distintas metodologías de desarrollo de *software*, para guiar en la elección de la metodología óptima en un proyecto específico.

Específicos

- Proporcionar una guía para el desarrollo de *software* que nos ayude a reducir tiempos y costos de implementación.
- Dar a conocer las distintas metodologías de desarrollo de *software*, ya que en la actualidad no son utilizadas debido al desconocimiento de las mismas.
- Realizar un análisis comparativo entre las distintas metodologías de desarrollo de *software*, para poder elegir la óptima de acuerdo a factores necesarios que deben de ser considerados en un proyecto de *software*.
- Proporcionar una matriz comparativa en donde cualquier estudiante o profesional de informática ubique un proyecto de *software* específico, y pueda deducir sin mayor problema la metodología de desarrollo óptima que se adapta a sus necesidades.

INTRODUCCIÓN

Debido a la rapidez con que la tecnología evoluciona, los sistemas de información se vuelven con el tiempo obsoletos debido a que no llegan a cubrir a cabalidad con los nuevos requerimientos. En el mejor de los casos, estos sistemas únicamente necesitan ser actualizados. Pero en el peor de los casos es necesario construir nuevamente el sistema con los nuevos requerimientos y utilizando la nueva tecnología.

Por lo general, al desarrollar un sistema de información se utiliza la metodología de análisis tradicional debido al desconocimiento de otras alternativas (prototipos, espiral, iterativo incremental, etc.), que podrían ser más útiles y adaptarse mejor a los requerimientos del sistema. Una determinación de requerimientos deficiente, da como resultado un sistema deficiente, además de incrementar los distintos costos: económicos, tiempo, recursos humanos, etc.

Esto se podría evitar, si antes de iniciar el proyecto se elige la metodología de desarrollo adecuada que satisfaga los nuevos requerimientos del sistema. Este documento pretende servir de ayuda en la elección de la metodología de desarrollo dando a conocer las ventajas y desventajas que tienen cada una de las metodologías de desarrollo (análisis tradicional, prototipos, espiral, iterativo incremental, etc.).

1. METODOLOGÍAS DE DESARROLLO DEL *SOFTWARE*

1.1 ¿Qué es *software*?

Una definición no muy formal de *software* podría ser la siguiente: son instrucciones de ordenador que cuando se ejecutan proporcionan la función y el comportamiento deseado, estructuras de datos que facilitan a los programas manipular de manera adecuada la información.

El *software* incluye no sólo los programas de ordenador, sino también las estructuras de datos que manejan esos programas y la documentación que se adjunta del proceso de desarrollo, mantenimiento y uso de dichos programas.

El hardware es algo físico, material, mientras que el *software* es inmaterial y por ello tiene unas características completamente distintas. Algunas de ellas pueden ser:

1.2 Características del *software*

El *software* es un elemento del sistema que es lógico, no es físico como el hardware. Por ello tiene unas características propias, como las citadas a continuación.

1.2.1 El *software* se desarrolla, no se fabrica

El proceso de desarrollo del *software,* independientemente de la metodología, sigue una serie de fases tales como análisis, diseño y desarrollo o construcción, obteniendo un buen producto final dependiendo de la calidad del diseño.

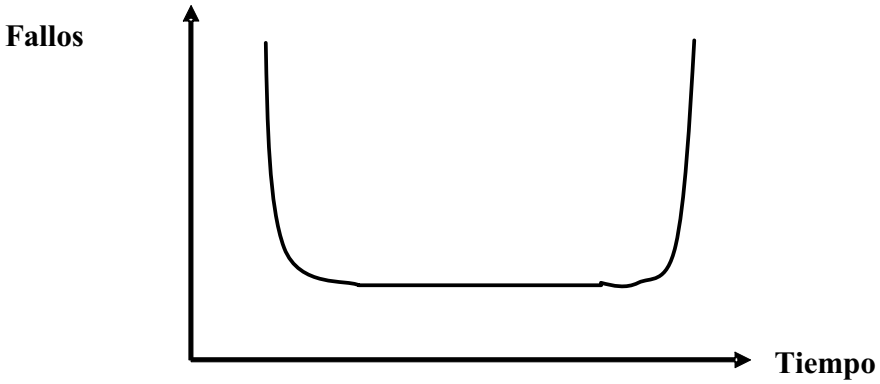
En el caso de producción de *hardware* a gran escala, el costo del producto depende del costo de los materiales empleados y del propio proceso de producción, no influyendo tanto en el costo las fases previas de ingeniería. En cambio, en el caso del *software*, el desarrollo es una más de las labores de ingeniería, y la producción a gran o pequeña escala no influye en el impacto que tiene la ingeniería en el costo, por ser un producto inmaterial. Por otro lado, el *software* no presenta problemas técnicos y no requiere un control de calidad individualizado, cosa que sí que ocurre con el *hardware*.

Los costos del *software* radican en el desarrollo de la ingeniería y no en la producción, y es ahí donde hay que incidir para reducir el costo final del producto.

1.2.2 El *software* no se estropea, pero se deteriora

Los agentes externos tales como la temperatura, humedad, tiempo, etc., no afectan la vida útil del *software*. Cosa que sí ocurre con el *hardware*. El *hardware* presenta muchos fallos al inicio de su fabricación, luego pasa por un punto estacionario y finalmente, por uno o más factores de los descritos anteriormente, comienza a desgastarse y la tasa de fallos comienza a incrementarse. Véase figura 1.

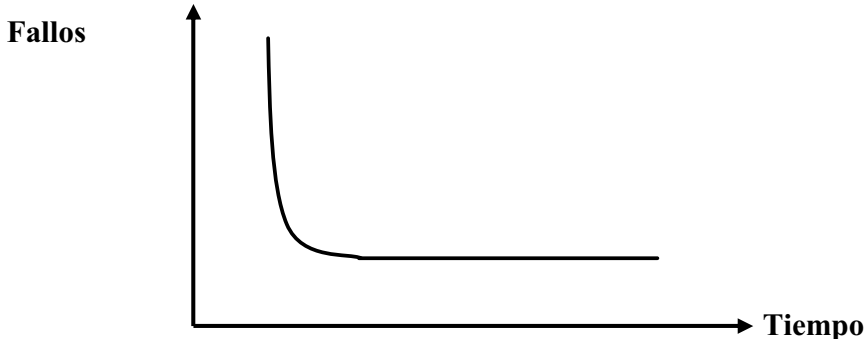
Figura 1. Curva de fallos del *hardware*



Fuente: Roger S. Pressman. **Ingeniería del *software*** . Página 10.

La curva del *software* (idealmente) tiende a aplanarse, pues los defectos no detectados en un inicio de su desarrollo harán que las primeras etapas de su vida fallen, pero conforme pasa el tiempo la curva se aplanará. Véase figura 2.

Figura 2. Curva ideal de fallos del *software*



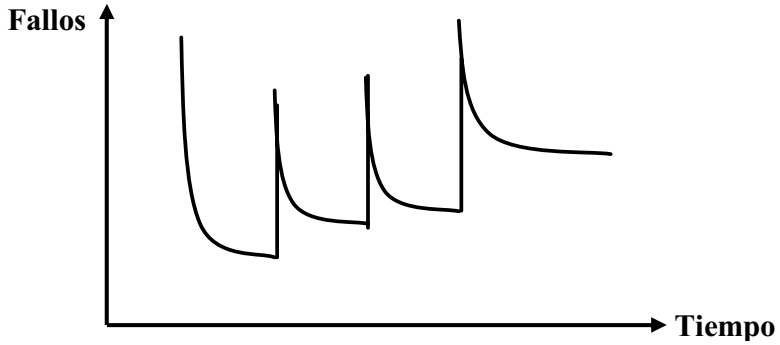
Fuente: Roger S. Pressman **Ingeniería del *software***. Página 10.

Esto no es más que una simplificación del modelo real de fallos de un producto de *software*. Porque durante su vida, el *software* va sufriendo cambios debido a su mantenimiento, el cual puede ser causado por distintos motivos tales como corrección de errores debido a cambios en los requisitos iniciales del producto.

Al realizar los cambios es posible que se produzcan nuevos errores, con lo cual se manifiestan picos en la curva de fallos. Estos errores pueden corregirse, pero los cambios posteriores hacen que el *software* se aleje cada vez más de las especificaciones iniciales de acuerdo a las cuales fue desarrollado, conteniendo cada vez más errores.

Además, a veces se solicita un nuevo cambio antes de haber corregido todos los errores producidos por el cambio anterior. Cada fallo en el *software* indica un fallo en el diseño o en el proceso por medio del cual se transformó el diseño a programa. La solución está en sustituir el diseño por otro así como el desarrollo del producto. Sin embargo no debemos olvidar que el coste para solucionar fallos en el *software* es más elevado que solucionar fallos en el *hardware*. Véase figura 3.

Figura 3. Curva real de fallos del software



Fuente: Roger S. Presuman. **Ingeniería del *software***. Página 10.

1.2.3 La mayoría de *software* se construye a la medida

En el caso del *hardware*, el diseño se realiza con componentes digitales existentes en catálogos que han sido probados por el fabricante y usuarios anteriores, teniendo especificaciones claras y bien definidas. Con el *software* no ocurre así. No existen catálogos de componentes o piezas, y aunque existen productos como sistemas operativos, editores de texto, hojas electrónicas, bases de datos, etc., se venden en ediciones o versiones, la gran mayoría del *software* se fabrica a medida, siendo su reutilización muy baja. Se puede comprar *software* ya desarrollado, pero como unidades o paquetes completos, no como componentes que pueden ser reutilizados para construir nuevos programas. Esto hace que el coste de ingeniería sobre el producto final sea elevado.

A lo largo de los años se ha intentado la reutilización del *software* pero tras muchos intentos ha obtenido poco éxito. La última tendencia es el uso de técnicas orientadas a objetos, que permiten la programación por especialización. Los objetos disponen de interfaces claras y los errores cometidos en su desarrollo pueden ser depurados sin que esto afecte a la corrección de otras partes del código y pueden ser heredados y modificados parcialmente, haciendo posible su reutilización aún en situaciones no contempladas en el diseño inicial.

1.3 Aplicaciones del *software*

El *software* puede aplicarse a numerosas situaciones del mundo real. En primer lugar, diremos que puede aplicarse a todos aquellos problemas para los que se ha establecido un conjunto de acciones que lleven a su resolución (algoritmo). En estos casos, usamos lenguajes para implementarlos. Sin embargo es difícil establecer categorías claramente definidas para las aplicaciones del *software*, porque cuanto más

complejo es el mismo, más complicado se hace establecer categorías genéricas. No obstante, se suele aceptar esta clasificación:

1.3.1 *Software* de sistemas

Está formado por aquellos programas cuyo fin es servir al desarrollo o al funcionamiento de otros programas, por ejemplo editores de texto, compiladores, sistemas operativos, entornos gráficos, programas de telecomunicaciones, etc. Pero todos ellos tienen un punto en común y es el interactuar muy próximo al *hardware*, ser utilizados por numerosos usuarios y, por ser programas de difusión, no están diseñados normalmente a medida. Esto permite un mayor diseño y optimización, pero también les obliga a ser muy fiables, cumpliendo estrictamente las especificaciones para las que fueron creados.

1.3.2 *Software* de tiempo real

Está formado por aquellos programas que miden, analizan y controlan los sucesos del mundo real a medida que estos ocurren, debiendo reaccionar de forma correcta a los estímulos de entrada en un tiempo máximo prefijado. Deben, por tanto, cumplir unos requisitos temporales muy estrictos, además de ser fiables y tolerantes a fallos. Por otro lado, no suelen ser muy complejos y precisan de poca interacción con el usuario. Se caracterizan por los siguientes componentes:

- **Adquisición de datos:** recoge y formatea los datos de entrada,
- **Análisis:** transforma la información dependiendo de la aplicación,
- **Control / salida:** responde a la entrada (entorno externo),
- **Monitorización:** coordina el resto de los componentes, de forma que pueda mantenerse,

- **Respuesta en tiempo real:** entre 1 milisegundo y 1 segundo.

1.3.3 *Software* de gestión

Este tipo de programas utiliza grandes cantidades de información almacenadas en bases de datos para poder facilitar las transacciones comerciales o la toma de decisiones. Además de las tareas convencionales de procesamiento de datos, en las que el tiempo de procesamiento no es crítico y los errores pueden ser corregidos posteriormente, incluyen programas interactivos que sirven de soporte a las transacciones comerciales.

1.3.4 *Software* científico y de ingeniería

Es otro de los campos clásicos de aplicación de la informática. Se encarga de realizar complejos cálculos sobre datos numéricos de todo tipo. En este caso, un requisito básico que deben cumplir es la corrección y exactitud de las operaciones que realizan. Este campo se ha ampliado últimamente con el desarrollo de los sistemas de diseño, ingeniería y fabricación asistida por computadoras (CAD, CAE y CAM), los simuladores gráficos y otras aplicaciones interactivas que lo acercan al *software* de tiempo real e incluso al de sistemas.

1.3.5 *Software* de computadoras personales

El uso de computadoras personales y de uso doméstico se ha extendido a lo largo de la última década. Aplicaciones típicas son los procesadores de texto, hojas de cálculo, bases de datos, aplicaciones gráficas, juegos, etc. Son productos de amplia difusión orientados a usuarios no profesionales, por lo que sus principales requisitos son la facilidad en el uso y el bajo costo.

1.3.6 *Software* empotrado

Es aquél que va instalado en la memoria ROM (*Read Only Memory*) o memoria de sólo lectura y se utiliza en productos industriales, por ejemplo la electrónica de consumo, dotando a estos productos de un grado de inteligencia cada vez mayor. Se aplica a todo tipo de productos, desde un vídeo doméstico hasta un misil con cabeza nuclear, pasando por sistemas de control de los automóviles, y realiza funciones muy diversas, desde complicados cálculos en tiempo real a sencillas interacciones con el usuario, facilitando el manejo del aparato que los incorpora. Comparten características con el *software* de sistemas, el de tiempo real, el de ingeniería y científico y el de computadoras personales.

1.3.7 *Software* de inteligencia artificial

El *software* basado en lenguajes por procedimientos es útil para realizar de forma rápida y fiable operaciones que para el ser humano son tediosas e incluso inabordables. Sin embargo, es difícil su aplicación a problemas que requieran funciones intelectuales más elevadas. Esta deficiencia trata de subsanarla el *software* de inteligencia artificial, basándose en el uso de lenguajes declarativos, sistemas expertos y redes neuronales. Esta clase de *software* permite aplicaciones muy diversas, pero en todas ellas encontramos algo en común: el objetivo es que el *software* determine cierta función cumpliendo, a su vez, una serie de requisitos (fiabilidad, corrección, respuesta en un tiempo determinado, facilidad de uso, bajo coste, etc.), a la hora de desarrollar el *software*. Ejemplos de *software*: Prolog, CLIPS, EHSIS.

1.4 ¿Qué es ISO 9000?

La Organización Internacional para la Estandarización (*The International Organization for Standardization*) es la agencia internacional especializada para la estandarización. Actualmente la forman las organizaciones nacionales de unos 130 países, una por cada país. Es una organización no gubernamental fundada en 1947.

El Instituto Nacional Americano de estándares (ANSI) es el miembro que representa a los Estados Unidos. La ISO está formada aproximadamente por 180 comités técnicos. Cada uno de estos es responsable por una de muchas áreas de especialización que van desde el asbesto hasta el zinc. El propósito de la ISO es promover el desarrollo de la estandarización y las actividades mundiales relacionadas para facilitar el intercambio internacional de productos y servicios, y para desarrollar la cooperación intelectual, científica, tecnológica y la actividad económica. Los resultados técnicos de la ISO son publicados como estándares internacionales.

El comité técnico ISO 176 (ISO / TC176) fue formado en 1979 para armonizar la siempre creciente actividad internacional en gerencia de calidad y estándares para aseguramiento de la calidad. El subcomité 1 fue establecido para determinar la terminología común. Este desarrolló ISO 8402, que es un vocabulario de calidad, publicado en 1986. El subcomité 2 fue establecido para desarrollar estándares de sistemas de calidad; el resultado fue la serie ISO 9000 publicada en 1987 y revisada en 1994.

La serie ISO 9000 es un grupo de 5 estándares internacionales individuales, pero relacionados, sobre gerencia de calidad y aseguramiento de la calidad. Los estándares son genéricos, no son específicos para un producto en particular. Pueden ser usados tanto por industrias manufactureras como empresas de servicios. Estos estándares fueron desarrollados para documentar efectivamente elementos del sistema de calidad en una

compañía. La serie ISO 9000 no especifica la tecnología que se debe usar para implementar elementos del sistema de calidad.

La ISO 9000 suministra al usuario guías para la selección y uso de las series ISO 9001, 9002, 9003 y 9004. Las ISO 9001, 9002, y 9003 son modelos de sistemas de calidad para el aseguramiento externo de la calidad. Estos tres modelos de hecho son sucesivos subgrupos de cada uno de ellos. La ISO 9001 es la más completa cubriendo diseño, manufactura, instalación y sistemas de servicios. El ISO 9002 cubre producción e instalación y el ISO 9003 cubre sólo la inspección final del producto y su prueba.

Estos tres modelos fueron desarrollados para uso en situaciones contractuales tales como aquellas entre cliente y proveedor. La ISO 9004 suministra guías para uso interno por un productor desarrollando su propio sistema de calidad para cumplir con las necesidades de los negocios y aprovechar las oportunidades. La selección sobre cual modelo implementar depende del alcance de sus operaciones. Por ejemplo, si usted diseña su propio servicio y producto, debe considerar la ISO 9001; si usted sólo fabrica (trabaja el diseño de otros) podría considerar la ISO 9002. Finalmente, si usted no diseña ni fabrica, debería considerar la ISO 9003.

1.4.1 ISO 9000 aplicado al *software*

En un área en donde la evolución tecnológica y las exigencias que ha traído la globalización, se ha hecho necesario desarrollar metodologías para asegurar la calidad de los productos de *software*. Es por eso que encontramos la Norma ISO 9000-3 y su derivación británica Tick IT; el "*Software Technology Diagnostic*" (STD), también del Reino Unido; el "*Capability Maturity Model*" (CMM), de Estados Unidos; la metodología europea *Bootstrap*; el modelo canadiense *Trillium*; el *HealthCheck* del *British Telecom*, y otras iniciativas que persiguen la mejora de los procesos que inciden directamente en este tipo de producto

La Serie ISO 9000, con un enfoque global y una aceptación mundial, le da un trato particular al soporte lógico con las siguientes normas:

- **ISO / IEC 9000 - 3:** Lineamientos para la aplicación de la Norma ISO 9001 en el desarrollo, suministro y mantenimiento del *software*.
- **ISO / IEC 9000 - 4:** Guía para la gestión de un programa de seguridad de funcionamiento. ISO / IEC 10007: directrices para la gestión de la configuración.

Y a pesar de que en algunos países todavía están en discusión para su aprobación, las siguientes normas también son muy importantes:

- **ISO / IEC 9126-1:** características de calidad del *software* y métricas (*Software Quality Characteristics and Metrics*).
- **ISO / IEC 12207:** proceso de ciclo de vida del *software* (*Software Life Cycle Processes*).
- **ISO / IEC 14102:** tecnologías de información - Pautas para la evaluación y selección de herramientas CASE (*Information technology - Guidelines for the evaluation and selection of* CASE *tools)*.
- **ISO / IEC 15026:** sistemas y niveles de integridad del *software* (*System and Software Integrity Levels*).
- **ISO / IEC 15271:** guía ISO / IEC para el proceso del ciclo de vida del *software* (*Guide to* ISO / IEC *Software Life Cycle Processes*).
- **ISO / IEC 15504:** valoración del proceso de *software* (*Software Process Assessment*).
- **ISO / IEC 15846:** dirección de configuración del *software* (*Software Configuration Management*).

También está el MIL-STD-498, estándar militar estadounidense para el desarrollo y documentación del *software*, el cual es aplicable para procesos del ciclo de vida del *software*. Es importante hacer notar que estas normas no recomiendan positiva ni negativamente el uso de una metodología en particular de desarrollo de *software*. La elección de una u otra es responsabilidad del gestor del proyecto, siempre que consiga dar satisfacción a los requisitos establecidos en el contrato.

1.4.1.1 Requisitos de ISO 9001

La Norma internacional ISO-9001 especifica los requisitos que debe cumplir un sistema de calidad cuando es necesario demostrar la capacidad de un proveedor para diseñar y suministrar productos conformes. La lista de esos 20 requisitos es la siguiente:

1. Responsabilidad de la dirección,
2. Sistema de calidad,
3. Revisión de contratos,
4. Control de diseño,
5. Control de documentos y de los datos,
6. Compras,
7. Control sobre los productos que suministramos,
8. Identificación y trazabilidad del producto,
9. Control de procesos,
10. Inspección y ensayos,
11. Control de equipos de inspección, medición y ensayos,
12. Estado de inspección y ensayo,
13. Control de productos no conformes,
14. Acciones correctivas y preventivas,
15. Manipulación, almacenamiento, embalaje, preservación y entrega,
16. Control de registros de calidad,

17. Auditorías internas,
18. Entrenamiento,
19. Servicio postventa,
20. Técnicas estadísticas.

La norma UNIT-ISO 9004 contempla otros dos elementos adicionales a los 20 anteriores:

- Consideraciones financieras de los sistemas de calidad y
- Seguridad de los productos

Pero aunque la calidad en *software* puede parecer un ideal inalcanzable, la empresa que logre armonizar la trilogía producto - proceso - persona. Cualquiera que sea la metodología que utilice, obtendrá la satisfacción del cliente

1.5 Metodologías para el desarrollo de *software*

Desarrollar un sistema de *software* no es algo que puede abordarse sin una preparación previa. El hecho de abordar un proyecto de desarrollo de *software* como cualquier otro ha llevado a una serie de problemas que limitan nuestra capacidad de aprovechar los recursos que el hardware pone a nuestra disposición.

Los problemas que a lo largo de los años han ido apareciendo no es algo que se va a solucionar en un corto espacio de tiempo, pero identificarlos y conocer sus causas es el único método que nos puede ayudar a solucionarlos.

Existen 3 problemas principales que afectan al desarrollo del *software*:

1. La planificación y estimación de costes son muy imprecisas,
2. La productividad del *software* es distinta a la demanda del mercado.
3. Baja calidad del *software* con relativa frecuencia.

Para solucionar estos problemas podemos combinar métodos completos para todas las fases del desarrollo del *software*; podemos utilizar herramientas para automatizar estos métodos y mejorar la calidad del *software*, en otras palabras, utilizar alguna metodología de desarrollo de *software* (Clásico, Prototipado, Espiral, Iterativo Incremental, etc.). La más apropiada para que minimice los problemas descritos anteriormente para el bienestar del proyecto de *software*.

2. CICLO DE VIDA CLÁSICO Y ESTRUCTURADO MODERNO

2.1 Ciclo de vida clásico

La noción de ciclo de desarrollo del *software* y las fases o etapas que lo componen, han variado ampliamente durante los años. Uno de los primeros modelos de ciclo de vida del desarrollo fue establecido por W. Royce en 1970 y es conocido como el "Modelo en cascada". Éste es el primero en establecer el proceso de desarrollo como la ejecución de un conjunto de actividades. Este es el más básico de todos los modelos, y sirve como bloque de construcción para los otros modelos de ciclo de desarrollo del *software*.

La visión del modelo es sencilla; 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 siguiente.

Las etapas o fases del modelo en cascada se detallan a continuación en los incisos 2.1.1 al 2.1.6.

2.1.1 Investigación preliminar

El objetivo de esta etapa es el de conocer con el mayor detalle posible toda la información necesaria acerca del proyecto. Para obtener toda la información podemos utilizar diferentes técnicas para determinar los requerimientos del sistema, algunas de ellas se detallan en los siguientes tres incisos.

2.1.1.1 La entrevista

La entrevista es una técnica de recopilación de información que permite la interacción con el usuario el cual establecerá con mayor libertad y confianza sus requerimientos y lo que espera obtener del sistema. Por medio de una plática directa con el cliente o usuarios, el encargado del proyecto conocerá como se presenta el problema y de los requerimientos y necesidades que son indispensables para iniciar el desarrollo.

Al final se elabora un reporte con la información que se logró recabar. Esta técnica permite tener una visión más clara y se conoce de manera más directa por el usuario los requerimientos y se tiene la oportunidad de tratar de ubicar al usuario en la realidad y hasta donde son factibles sus expectativas.

Para la realización de una entrevista deben planearse con anterioridad los siguientes pasos:

- Determinar cuáles son los objetivos de realizar la entrevista.
- Seleccionar el personal clave para la entrevista.
- Decidir que estructura de entrevista (piramidal, embudo, diamante) se va a utilizar.

Para seguir el desarrollo adecuado de una entrevista, se deben tomar en cuenta los siguientes aspectos:

- Si se pierde el contacto visual con el entrevistado, se puede perder su atención.
- Si se desechan los detalles proporcionados, se pierden características del sistema.

- Siempre utilizar un lenguaje sencillo y claro, para tener una mejor comunicación con el entrevistado.
- Si la entrevista excede el tiempo estipulado, esta puede volverse tediosa y aburrida.

Para determinar la estructura de la entrevista debe conocer las estructuras existentes y adecuarla a la necesidad de información requerida, por ejemplo:

- **Piramidal:** se empieza preguntando de lo específico hasta terminar en lo general.
- **Embudo:** Se empieza con una plática general para terminar con preguntas específicas.
- **Diamante:** Esta es una combinación de las anteriores, y va de la pregunta más general a la más específica y retoma nuevamente la parte general hasta llegar a la máxima generalización.

2.1.1.2 Cuestionarios

Esta técnica consiste en elaborar preguntas más concretas y específicas para obtener información de requerimientos del cliente o usuarios. La aplicación de estas preguntas se hace de manera escrita, lo que hace más limitada la interacción y comunicación del encargado del sistema y los usuarios. Las posibles maneras de elaborar un cuestionario son las siguientes:

- **Cuestionario de preguntas abiertas:** se le da la libertad al usuario de expresar lo que considere más apropiado.
- **Cuestionario de preguntas cerradas:** las respuestas están limitadas a un “sí” o un “no”, "verdadero" o "falso".

- **Cuestionario de opción múltiple:** se proporciona una serie de posibles respuestas, de las cuales el usuario tiene que elegir, entre las más probables que sean las correctas, la que más se acerque a lo que considere la respuesta correcta.

2.1.1.3 Observación

Esta técnica consiste en que el encargado del proyecto hace una visita a las áreas donde se utilizará el sistema, para saber cómo se realizan las operaciones. Hará una revisión del sistema actual y de las operaciones que se realizan y las necesidades que se cubren. Con esto elaborará un reporte con los datos y hechos más sobresalientes observados en su recorrido. Por medio de esta técnica se logrará tener una visión más amplia de lo que se requiere y de ubicarse en la realidad de la situación actual.

2.1.2 Análisis del sistema

En la etapa de análisis se establecen los requerimientos del producto que se desea desarrollar. Éstos consisten usualmente en los servicios que deben proveer, limitaciones y metas del *software*. Una vez establecidos los requerimientos, éstos deben ser definidos de una manera apropiada para ser útiles en la siguiente etapa.

Esta etapa incluye también un estudio de la factibilidad y viabilidad del proyecto con el fin de determinar la conveniencia de la puesta en marcha del proceso de desarrollo. Puede ser tomada como la concepción de un producto de *software* y ser vista como el comienzo del ciclo de vida.

2.1.3 Diseño del sistema

Una vez que conocemos qué hay que hacer, en esta fase se determina cómo se va a hacer. El diseño del *software* es un proceso que se centra en cuatro atributos diferentes de los programas:

- Diseño de la estructura de datos. Define la manera en que se va a almacenar la información del sistema, es decir, el modelo de datos, modelo entidad relación, diagrama de clases, según el enfoque utilizado.
- Diseño de la arquitectura del *software*. Define las relaciones existentes entre las entidades o clases del modelo de datos. Relaciones tales como de uno a uno, uno a muchos, muchos a muchos si es un modelo de entidad relación, o relaciones de generalización, asociación, dependencia y agregación o composición si se trata de un diagrama de clases.
- Diseño detallado del proceso y / o procedimientos. Define todas las funciones, procedimientos, métodos que dan solución a una parte específica del sistema. La integración de todos los procesos y / o procedimientos dan solución al problema para el cual fueron creados.
- Diseño de la interfaz. Define la manera en que se comunica el usuario con el sistema.

El proceso de diseño representa los requerimientos en una forma que permita la codificación del producto, además de una evaluación de la calidad previa a la etapa de codificación. Algunos criterios para evaluar la presentación del diseño son las siguientes:

a) Debe presentar una organización jerárquica.
b) Debe ser modular, es decir debe, de poseer una funcionalidad independiente.

c) Debe contener abstracciones de datos, es decir, partir de lo general a lo específico.
d) Debe producir un diseño fácil de entender y comprender.

2.1.4 Construcción o codificación y documentación interna

2.1.4.1 Construcción o codificación

Etapa en la cual son creados los programas, es el traslado del análisis y diseño a un lenguaje de programación. Es importante codificar el programa respetando los estándares de programación definidos por el gestor del proyecto, por ejemplo, elección del nombre y definición de las variables. Es recomendable utilizar alguna notación para identificar los diferentes tipos de datos; posiblemente notación húngara que consiste en anteponer un prefijo a cada variable para identificar su tipo de dato (entero, real, cadena, etc.).

También es muy importante utilizar nombres significativos para las variables y constantes, y la estructura del código debe de mantener una tabulación adecuada para facilitar su visualización y comprender mejor su estructura.

2.1.4.2 Documentación interna

La documentación interna se refiere a todas aquellas anotaciones y / o comentarios contenidos en el programa fuente. Provee información general del programa, lo mismo que información relevante de los módulos. Los diferentes tipos de documentación interna comúnmente utilizados son los que a continuación se enumeran:

a) Comentarios de cabecera de programa, contienen información general del programa tal como:

- Nombre del programa
- Descripción general del programa
- Fecha de inicio
- Fecha de finalización
- Programador (es)
- Referencias externas
- Archivos usados

b) Comentarios de cabecera de módulo, contienen información sobre el procedimiento, función o subrutina individual.

- Nombre del módulo
- Descripción específica del módulo
- Fecha de inicio
- Fecha de finalización
- Fecha de la última revisión y / o modificación
- Programador (es)

c) Comentarios de línea, proporcionan información detallada sobre algún bloque corto de instrucciones contenidas en un módulo, explicando breve y consistentemente la funcionalidad del mismo.

2.1.5 Verificación o pruebas

La prueba del *software* es un elemento crítico para la garantía de calidad del *software* y representa una revisión final de las especificaciones del diseño y de la codificación. Una vez finalizada la codificación, comienza el proceso de pruebas a cada módulo del programa.

Probar un programa es ejecutarlo con la intención de encontrarle fallos. El proceso de verificación se centra en dos puntos principales: las lógicas internas del *software*; y las de funcionalidad externa, es decir se solucionan errores de comportamiento del *software* y se asegura que las entradas definidas producen resultados reales que coinciden con los requerimientos especificados.

2.1.5.1 Estrategias de prueba

Una estrategia de prueba es un enfoque general al proceso de prueba más que un método de llevar a cabo pruebas particulares sobre sistemas o componentes. Se pueden adoptar diferentes estrategias dependiendo del tipo de sistema a probar y el proceso de desarrollo utilizado. A continuación se detallan diferentes estrategias de prueba.

a) **Prueba descendente** ***(top down)****:* comienza con los componentes fundamentales y avanza hacia abajo.
   - Se comienza por los niveles más altos del sistema y avanza hacia abajo.
   - Se utiliza junto con el desarrollo descendente.
   - Encuentra errores de arquitectura.
   - Necesita disponer de la infraestructura del sistema antes de realizar la prueba.

b) **Prueba ascendente *(bottom up):*** comienza con los componentes fundamentales y se avanza hacia arriba.
    - Necesaria para componentes críticos de la infraestructura.
    - Comienza en los niveles inferiores y avanza hacia arriba.
    - Necesita controladores de pruebas para implementarse.
    - No encuentra los problemas de diseño hasta muy avanzado el proceso.
    - Apropiado para sistemas orientados a objetos.

c) **Prueba de hebras:** se utiliza para sistemas con múltiples procesos donde el procesamiento de una transacción sigue su camino a través de estos procesos.
    - Adecuada para sistemas de tiempo real y orientado a objetos.
    - Basada en la prueba de una operación que incluye una secuencia de pasos de procesamiento que avanza a través del sistema.
    - Comienza con hebras de eventos simples y después con las de eventos múltiples.
    - Es imposible una prueba completa por el gran número de combinaciones de eventos.

d) **Pruebas de tensión:** consisten en forzar el sistema yendo más allá de los límites especificados y probando a partir de esto cómo se desenvuelve el sistema en situaciones de sobrecarga.
    - Ejercita el sistema bajo su máxima carga de trabajo. Normalmente hace que afloren los defectos del sistema.
    - Los sistemas no deberían fallar estrepitosamente.
    - Especialmente indicado para sistemas distribuidos.

Los sistemas grandes se prueban normalmente utilizando una mezcla de estas estrategias de prueba más que con una sola de ellas. Pueden ser necesarias diferentes

estrategias para partes distintas del sistema y diferentes etapas del proceso de prueba. Siempre se deben de realizar pruebas al *software* por las siguientes razones:

1. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Si la prueba se lleva a cabo con éxito, de acuerdo con los objetivos anteriormente escritos, se descubrirán errores en el *software*. Como una ventaja adicional las pruebas demuestra hasta qué punto las funciones del *software* parecen funcionar de acuerdo con las especificaciones y alcanzan los requisitos de rendimiento establecidos.

Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del *software* y, de alguna manera indican la calidad del *software* como un todo.

Es importante tener en mente lo siguiente cuando se lleva a cabo una prueba: no puede garantizar la ausencia de defectos, sólo puede demostrar que existen defectos en el *software*. Cualquier producto de ingeniería y de muchos otros campos puede ser probado de una de dos formas:

a) Conociendo la función especifica para la que fue diseñado el producto se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y,
b) 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.

El primer enfoque de prueba se denomina pruebas de caja negra y la segunda prueba de la caja blanca.

2.1.5.2 Pruebas de la caja blanca

Mediante los métodos de prueba de la caja blanca, el ingeniero del *software* puede obtener casos de prueba que:

- Garanticen que se ejercitan por lo menos una vez todos los caminos independientes de cada módulo.
- Ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa.
- Ejecuten todos los bucles en sus límites y con sus limites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.

2.1.5.2.1 Pruebas del camino básico

El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedural y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.

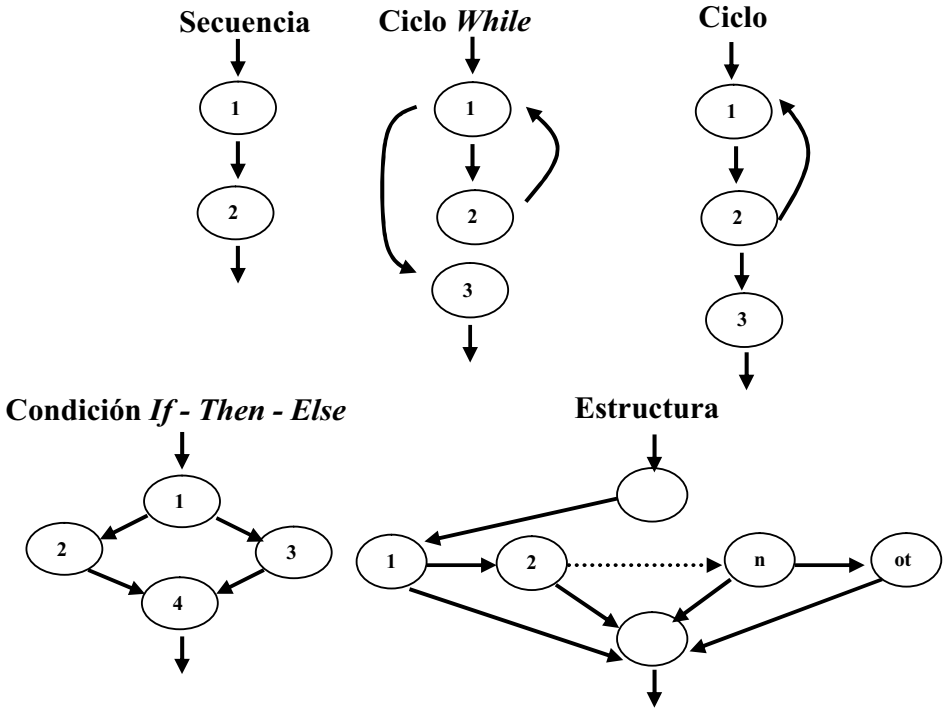
Los casos de prueba derivados del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa. Los pasos para realizar esta prueba son los siguientes:

1. Dibujar el grafo de flujo, partiendo del diseño o del código del procedimiento o función.
2. Obtener la complejidad ciclomática del grafo de flujo a partir del número de regiones en que el grafo divide al plano.
3. Establecer un conjunto básico de caminos linealmente independientes.
4. Determinar los casos de prueba que permitan la ejecución de cada camino del conjunto anterior.
5. Ejecutar cada caso de prueba y comparar con los resultados esperados.

Para construir el grafo de flujo hay que transformar las instrucciones del código de programación en subgrafos de flujo adecuados. Las instrucciones que debemos transformar son los ciclos iterativos como *for, while, repeat*, asignación de valores y toma de decisiones como el *if* y *case* respectivamente.

Estas transformaciones las llevaremos a cabo de acuerdo a las estructuras mostradas en la siguiente figura:

Figura 4. Notación de grafos de flujo



Fuente: Roger S. Pressman. **Ingeniería del *software***. Página 306.

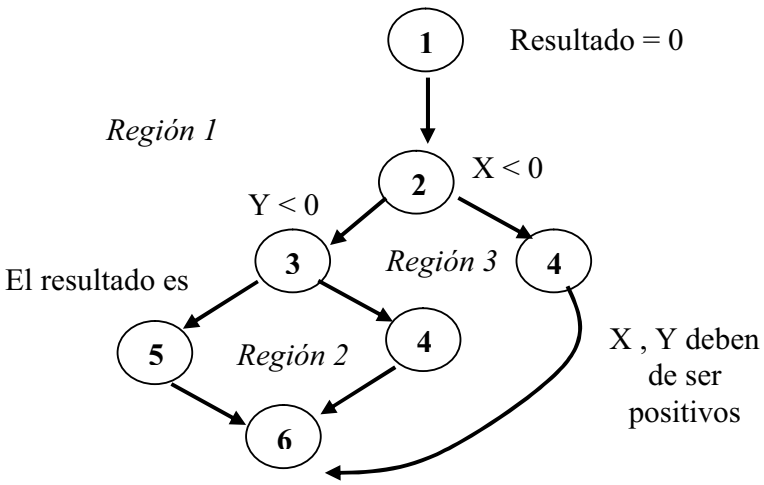
Ejemplo 1:

```
void ImprimeMedia(float X, float Y)
{
        float resultado=0;
        if ((X<0) || (Y<0))
        {
                printf("%2.2f y %2.2f deben de ser positivos",X,Y);
        } else {
                Resultado = (X + Y) / 2;
                printf("El resultado es %2.2f\n",resultado);
        }
}
```

Solución:

1. Obtener el grafo de flujo partiendo del código fuente

Figura 5. Ejemplo de grafo de flujo



2. Obtener la complejidad ciclomática del grafo

   La complejidad ciclomática la podemos calcular de las siguientes formas:

   - V(G) = # de regiones del grafo,
   - V(G) = A - N + 2 donde A = # de aristas, N = # de nodos
   - V(G) = 3
   - V(G) = 8 - 7 + 2 → V(G) = 3

3. Determinar el conjunto de caminos básicos linealmente independientes,

   - Camino # 1: 1 - 2 - 3 - 5 - 6
   - Camino # 2: 1 - 2 - 3 - 4 - 6
   - Camino # 3: 1 - 2 - 4 - 6

4. Determinar los casos de prueba que permitan la ejecución de cada camino del conjunto anterior y,

    a) Elegir X e Y tales que se cumpla X mayor o igual a 0, Y mayor o igual a 0
    b) Elegir X e Y tales que se cumpla X mayor o igual a 0, Y menor que 0
    c) Elegir X e Y tales que se cumpla X sea menor que 0, Y mayor que 0

5. Ejecutar cada caso de prueba y comparar con los resultados esperados

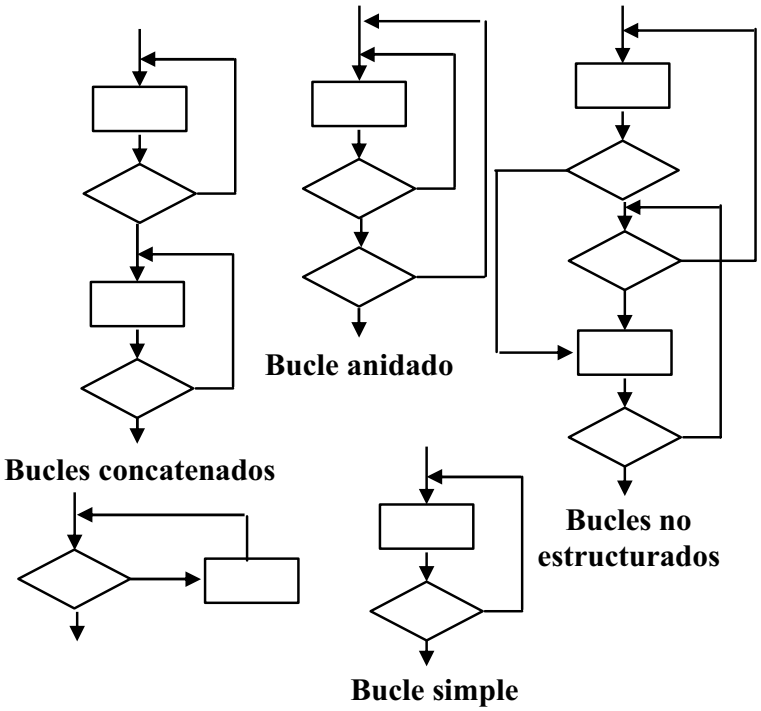
    a) Valores iniciales X = 4, Y = 2
        Salida → El resultado es 6
    b) Valores iniciales X = 0, Y = -1
        Salida → X, Y deben ser positivos
    c) Valores iniciales X = -1, Y = 5
        Salida → X, Y deben ser positivos

2.1.5.2.2 Pruebas de bucles

Esta prueba se centra en la validez de las construcciones de bucles. Intenta descubrir errores de inicialización, errores de indexación o de incremento y errores en los límites de los mismos.

Los tipos de bucles que se validan son los siguientes: simples, anidados, concatenados y no estructurados, y se representan de a cuerdo a las estructuras mostradas en la siguiente figura:

Figura 6. Notación de bucles



Fuente: Roger S. Pressman. **Ingeniería del *software***. Página 314.

Bucles simples: se debe de aplicar el siguiente conjunto de pruebas, en donde n es el número máximo de pasos permitidos para el bucle.

1. Saltar totalmente el bucle,
2. Pasar una sola vez por el bucle,
3. Pasar dos veces por el bucle,
4. Hacer m pasos por el bucle con m < n,
5. Hacer n - 1, n y n + 1 pasos por el bucle.

Bucles anidados: las pruebas para este tipo de bucles son los que a continuación se detallan:

1. Comenzar por el bucle más interno. Colocar todos los demás bucles en sus valores mínimos,
2. Realizar las pruebas de bucles simples con el bucle más interno, mientras se mantienen los bucles externos con los valores mínimos. Agregar otras pruebas para los valores fuera de rango o para valores excluidos,
3. Avanzar hacia fuera, llevando a cabo las pruebas para el siguiente bucle, pero manteniendo los demás bucles externos en sus valores mínimos y los demás bucles anidados con sus valores típicos,
4. Continuar este proceso hasta que se hayan probado todos los bucles.

Bucles concatenados: se pueden evaluar como si fueran bucles simples, siempre y cuando cada uno sea independiente del resto. Cuando los bucles no son independientes, se recomienda evaluarlos como si se fueran anidados.

Bucles no estructurados: en la medida de lo posible, esta clase de bucles se deben reestructurar para que se ajusten a las construcciones de la programación estructurada.

2.1.5.3 Pruebas de la caja negra

Este tipo de pruebas se centra en la funcionalidad del *software*. Y el objetivo es tratar de encontrar errores en:

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores de estructura de datos.
- Errores de rendimiento.

- Errores de inicialización y finalización.

2.1.5.3.1 Método de la partición equivalente

Este método trata de obtener un pequeño subconjunto de todas las posibles entradas con la mayor probabilidad de encontrar todos los errores.

Todo caso de prueba bien seleccionado debe de cumplir con los siguientes requisitos:

a) Reducir, en un coeficiente mayor que uno, el número de casos de prueba adicionales que se deben realizar para alcanzar una prueba razonable. Es decir que cada caso de prueba debe incluir tantas condiciones de entrada distintas como sea posible.
b) Cubrir a un gran conjunto de posibles casos de prueba. Es decir, nos dice algo sobre la presencia o ausencia de un error asociado solamente con la prueba en particular que se encuentra disponible. Por lo tanto, debemos tratar de dividir el dominio de entrada de un programa en un número finito de clases de equivalencia, para que se pueda asumir de forma razonable un valor representativo de cada clase es equivalente a probar algún otro valor.

Estas dos propiedades forman la metodología de la partición equivalente. El segundo requisito se usa para desarrollar un conjunto de condiciones posibles para ser evaluadas. El primer requisito se usa después para desarrollar un conjunto mínimo de casos de prueba que cubran esas condiciones.

Los pasos para realizar la prueba del método de la partición equivalente son los siguientes:

1. **Identificación de las clases de equivalencia:** se identifican dividiendo en dos o más grupos cada condición de entrada.

   Se usa la siguiente tabla:

Tabla I. Partición equivalente

CONDICIONES DE ENTRADA	CLASES DE EQUIVALENCIA VÁLIDAS	CLASES DE EQUIVALENCIA NO VÁLIDAS

Se identifican siempre dos tipos de clases equivalencia:

- Clases de equivalencia válidas.
- Clases de equivalencia inválidas que son las entradas erróneas.

Directrices para definir las clases de equivalencia:

a) Si una condición de entrada especifica un rango de valores, se define una clase de equivalencia válida y dos no válidas.
b) Si una condición de entrada especifica una situación, se define una clase de equivalencia válida y otra no-válida.
c) Si una condición de entrada especifica un conjunto de entradas y conocemos que cada valor va a tener un tratamiento diferente en el programa, se define una clase de equivalencia válida y una clase de equivalencia no válida.
d) Si una condición de entrada especifica el número de valores, identificar una clase de equivalencia válida y dos no válidas.

e) Si tenemos una razón para creer que los elementos de una clase de equivalencia no se tratan de la misma manera en el programa, dividiremos la clase de equivalencia en clases más pequeñas.

2. **Identificación de los casos de prueba:** los pasos para definir una caso de prueba son los siguientes:

   a) Se asigna un número único a cada clase de equivalencia.
   b) Se escriben casos de prueba hasta que sean cubiertas todas las clases de equivalencia válidas, intentando cubrir en cada caso tantas clases de equivalencia como sea posible.
   c) Se escriben casos de prueba hasta que sean cubiertas todas las clases de equivalencia no válidas cubriendo en cada caso una, y sólo una, clase de equivalencia aún no-cubierta.

La razón principal por la que las clases de equivalencia no válidas se cubren de forma individual se debe a que al detectar una entrada errónea seguramente no se chequea la siguiente entrada.

Ejemplo 2:

Un programa solicita 3 valores enteros. Los 3 valores son interpretados como la longitud de los 3 lados de un triángulo. El programa muestra un mensaje en pantalla indicando si el triángulo es escaleno, isósceles o equilátero.

Solución:

1. Identificando las clases de equivalencia:

Tabla II. Solución ejemplo 2

CONDICIONES DE ENTRADA	CLASES DE EQUIVALENCIA VÁLIDAS	CLASES DE EQUIVALENCIA NO VÁLIDAS
El número de valor es	[1] 3	[2] < 3 (3) > 3
Los valores son	[4] enteros	[5] cualquier otro valor
Valores mayores que 0	[6] todos mayores que 0	[7] alguno menor o igual a 0

2. Identificación de los casos de prueba:

[1] 3 - 5 - 4 (encierra todas las clases válidas [1] - [4] - [6])
[2] 2 - 5 (2 datos)
[3] 2 - 5 - 8 - 9 (4 datos)
[4] 2 - 5 - 'C' (dato no entero)
[5] -2 - 2 - 5 (algún dato menor o igual a 0)

Un ejemplo de error no detectado es el siguiente: 2 - 3 - 5, devuelve el mensaje "triángulo escaleno", cuando realmente no se puede formar el triángulo por la siguiente condición a + b > c.

2.1.5.3.2 Análisis de valores límite

Una condición límite es aquella situación que se dan cuando se introduce un valor que está justo en el límite de las clases de equivalencia de entrada y de salida.

Las principales diferencias entre éste método y el anterior son las siguientes:

a) No se selecciona un elemento representativo de una clase de equivalencia sino uno o más elementos tal que se prueben los extremos de esa clase de equivalencia.
b) Prestan atención no sólo a las condiciones de entrada (espacio de entradas) sino también al espacio de resultados.

Para la aplicación del método se requiere creatividad y cierto nivel de especialización en el problema que se trata, los criterios a tomar en cuenta son los siguientes:

a) Si una condición de entrada especifica un rango de valores, se escriben casos de prueba para los límites del rango y casos de prueba inválidos para situaciones justo fuera de los límites.
b) Si una condición de entrada especifica un número de valores, se escriben casos de prueba para el número máximo y mínimo de valores y otros para los valores justo por encima y justo por debajo.
c) Se utiliza los criterios anteriores para cada condición de salida.
d) Si en la entrada o salida de un programa se encuentra en un conjunto ordenado se presta atención al primero y al último elemento del conjunto.

Ejemplo 3:

Un programa solicita 3 valores enteros. Los 3 valores son interpretados como la longitud de los 3 lados de un triángulo. El programa muestra un mensaje en pantalla indicando si el triángulo es escaleno, isósceles o equilátero.

Solución:

Además de identificar en la tabla las condiciones de entrada, debemos considerar las condiciones de salida. Además, cuando tomemos los casos de prueba debemos recordar que elegiremos los valores extremos de las clases de equivalencia obtenidas.

Tabla III. Solución ejemplo 3

CONDICIONES DE ENTRADA / SALIDA	CLASES DE EQUIVALENCIA VÁLIDAS	CLASES DE EQUIVALENCIA NO VÁLIDAS
El número de valor es	[1] 3	[2] < 3 y [3] >3
Los valores son	[4] enteros	[5] cualquier otro dato
Valores mayores que 0	[6] todos mayor que 0	[7] alguno <= 0
Número de resultados	[8] 1	[9] < 1 y [10] > 1
Tipo de salida	[11] cadena	[12] cualquier otro dato
Relación entre longitudes	[13] a + b > c	[14] a + b <= c
Cuando a = b = c	[15] Equilátero	[12] cualquier otro dato
Cuando dos lados son iguales	[17] Isósceles	[12] cualquier otro dato
Cuando los tres lados son distintos	[19] Escaleno	[12] cualquier otro dato

Se puede observar que no es posible incluir todas las clases de equivalencia válidas en un solo caso de prueba. Sin embargo, se intenta incluir todas las clases de equivalencia válidas en el mínimo número de casos de prueba y cada clase de equivalencia no válida supone 1 o más casos de prueba separados.

Casos de prueba:

Tabla IV. Casos de prueba

CASO DE PRUEBA	ENTRADA	SALIDA	OBSERVACIÓN
A	1, 1, 1	Equilátero	Clases de equivalencia válidas 1, 4, 6, 8, 11, 13, 15, 17 y 19
B	1, 2, 2	Isósceles	Clases de equivalencia válidas 1, 4, 6, 8, 11, 13 y 17
C	2, 3, 4	Escaleno	Clases de equivalencia válidas 1, 4, 6, 8, 11, 13 y 19
D	1, 1	Indefinida	Clase de equivalencia no válida 2
E	2, 5, 'C'	Indefinida	Clase de equivalencia no válida 5
F	-2, 2, 5	No es triángulo	Clase de equivalencia no válida 7

2.1.6 Mantenimiento y documentación externa

2.1.6.1 Mantenimiento

Esta etapa consiste en la corrección de errores que no fueron previamente detectados, mejoras funcionales y otros tipos de soporte. La etapa de mantenimiento es parte del ciclo de vida del producto de *software* y no pertenece estrictamente al desarrollo. Sin embargo, mejoras y correcciones pueden ser consideradas como parte del desarrollo.

Las actividades de análisis, durante el mantenimiento, implican la comprensión del alcance y efecto de una modificación deseada, además de las restricciones para hacer la modificación. Existen actividades que son llevadas a cabo en cada una de las etapas del desarrollo del *software*. Éstas son documentación, verificación y administración. La

documentación es intrínseca al modelo cascada, puesto que la mayoría de las salidas que arrojan las etapas son documentos.

El mantenimiento se puede clasificar en las siguientes categorías:

- **Mantenimiento de perfeccionamiento:** se debe a cambios solicitados por el usuario o por el programador del sistema. Se busca mejorar el rendimiento.
- **Mantenimiento adaptativo:** se debe a cambios en el ambiente del programa. Cambios en la tecnología o en el sistema operativo. Por ejemplo, reemplazo de formas de ingreso con controles más adecuados para la captura de información, que la versión anterior del lenguaje de programación no tenía disponibles en ese momento.
- **Mantenimiento correctivo:** es la corrección de errores del sistema no descubiertos en el proceso de prueba.
- **Mantenimiento preventivo:** lo que se hace para prevenir el caos del *software* como consecuencia de las modificaciones sucesivas. Este tipo de mantenimiento va en paralelo con los mantenimientos correctivo, adaptativo y perfectivo. Se deben definir procedimientos que hay que realizar para prevenir tal caos.

2.1.6.2 Documentación externa

Se entiende por documentación externa toda aquella documentación que contiene información técnica referente al proyecto, y que se le proporciona al cliente como material de apoyo en el aprendizaje y / o configuración del sistema.

Los distintos documentos que forman parte de la documentación externa son lo que a continuación se detallan:

1. **Documentación del sistema:** describe los aspectos generales del sistema, tales como la problemática que soluciona el sistema, arquitectura del sistema, descripción funcional, beneficios de la solución, etc.

2. **El manual de instalación:** debe especificar el equipo donde se hará la instalación (configuración mínima), los archivos permanentes, cómo iniciar el sistema y los archivos dependientes de la configuración que se modifican para adecuar el sistema a una aplicación particular.

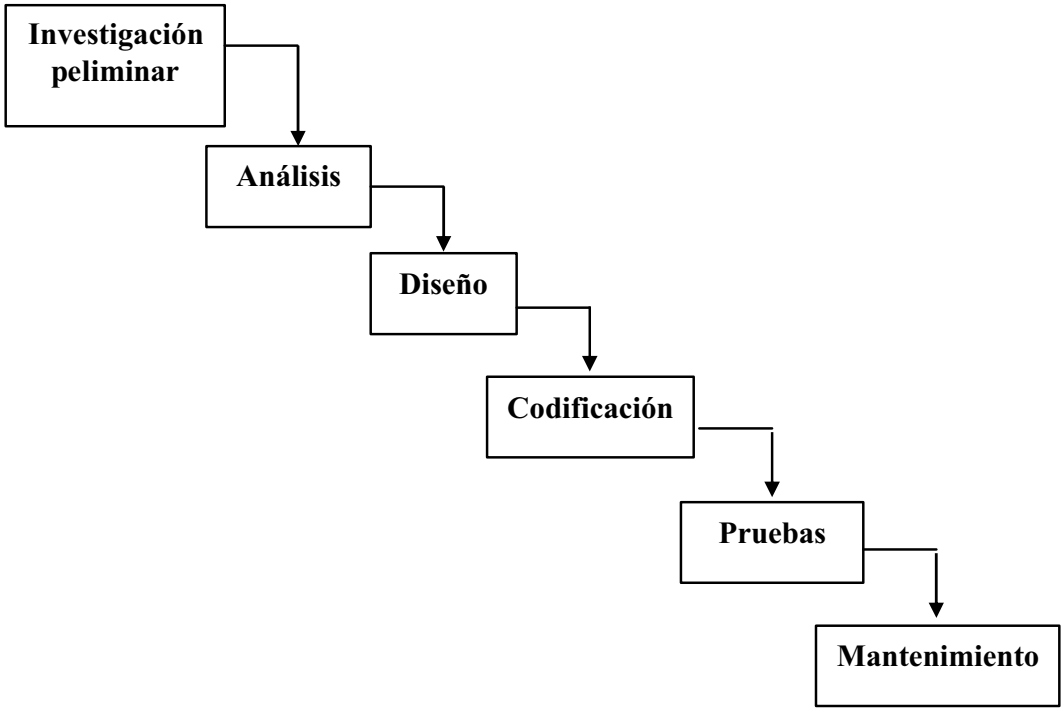
3. **Documentación del usuario:** son los documentos relacionados con las funciones del sistema. Debe proporcionar una visión inicial precisa del sistema. Contenido de la documentación del usuario.

    - Una descripción funcional de lo que puede hacer el sistema.
    - Cómo instalar el sistema y adecuarlo a otras configuraciones particulares del *hardware*.
    - Un manual de introducción que explique, en forma sencilla cómo iniciarse en el sistema.
    - Un manual de referencia que describa con detalle las ventajas del sistema, disponibles para el usuario y cómo se pueden usar.
    - Una guía del operador (si se requiere operador del sistema), que explique cómo debe reaccionar éste ante situaciones surgidas mientras el sistema se encuentra en uso.

2.1.7 Esquema del modelo en cascada

La siguiente grafica se muestra el esquema del modelo en cascada, obsérvese la secuencia de las distintas etapas para obtener como resultado el sistema final.

Figura 7. Esquema del modelo en cascada



Fuente: **Ciclo vida *software***. http://mural.uv.es/givaro/modulo/Ciclo.htm. (20/08/2002)

2.2 Análisis estructurado moderno

Antes de explicar en qué consiste el análisis estructurado, veamos algunas definiciones simples de análisis:

- Estudio realizado para separar las distintas partes de un todo.
- El análisis es el estudio de un problema, antes de realizar alguna acción.

Aunque la primera es una definición completamente válida, es más tradicional, mientras que la segunda se aproxima más a nuestros intereses, es decir, al análisis con respecto al proceso de desarrollo de *software*.

Se han creado distintas maneras de realizar este análisis, pues aunque en teoría es un proceso similar al del análisis en cualquier otro campo ya sea abstracto o concreto. En los sistemas de información se deben tener muchas consideraciones con respecto al trabajo que se desea realizar. Como respuesta a esta necesidad en el año de 1979 surgió el método de análisis estructurado, creado por Tom DeMarco.

Su principal objetivo es el proveer a los constructores del sistema el Documento de Especificación Estructurada (DEE), el cual es el resultado del uso de distintas herramientas, principalmente los Diagramas de Flujo de Datos (DFD) y los Diccionarios de Datos (DD). El DEE se complementa mediante el uso de Lenguaje Estructurado, Tablas de Decisión y Árboles de Decisión.

Cabe notar que hay áreas dentro del campo del análisis que no están incluidas en el análisis estructurado, como pueden ser un análisis de costo-beneficio, de desempeño del sistema, de selección de equipo o de políticas de la empresa. Como se expuso anteriormente, DFD y los DD constituyen la mayor parte del cuerpo del DEE. A continuación se definen y describen ambos.

2.2.1 Diagramas de flujo de datos

En pocas palabras, los DFD son representaciones gráficas de los procesos de un sistema, incluidas sus entradas y salidas de información. Estos están enfocados en el flujo de los datos del sistema y son representados en forma de red. Para representar el sistema gráficamente se utilizan cuatro elementos principales, que deben representar en el diagrama cada componente a considerar del sistema. Estos son los flujos, los procesos, los archivos y las fuentes de datos o depósitos. A continuación se definirá cada componente.

2.2.1.1 Flujo

Los flujos representan la comunicación entre el resto de los componentes de un diagrama y, aunque por lo general están conectados solamente a dos de ellos, pueden divergir o converger, es decir, pueden separarse o unirse dos o más. El nombre del flujo nos da una idea de los datos que se traspasan de un elemento a otro, pero la definición correcta y única de ello se encuentra en el Diccionario de Datos. Los flujos se representan mediante una flecha unidireccional.

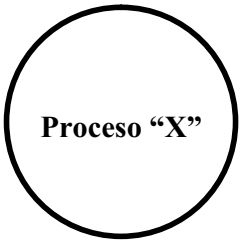
Figura 8. Representación de un flujo de datos

Fuente: Kennet E. Kendal y Julie E. Kendall. **Análisis y diseño de sistemas**. Página 231.

2.2.1.2 Proceso

Se pueden considerar como la parte principal de un diagrama. Representan las acciones u operaciones que se llevan a cabo en el sistema, de acuerdo a las entradas que reciba, para producir salidas esperadas. Éstos se dibujan con una burbuja.

Figura 9. Representación de un proceso

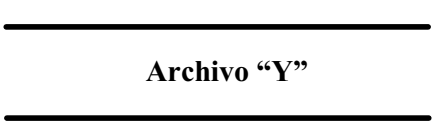


Fuente: Kennet E. Kendal y Julie E. Kendall. **Análisis y diseño de sistemas**. Página 231.

2.2.1.3 Archivo

Como archivo podemos tomar cualquier dispositivo de almacenamiento de información relevante al sistema, que guarde de manera temporal o permanente, en memoria, disco duro, cinta magnética, etc. De los archivos se pueden introducir y extraer datos. La notación para los archivos consta de un título sobre una línea recta o entre dos líneas horizontales.

Figura 10. Representación de un archivo

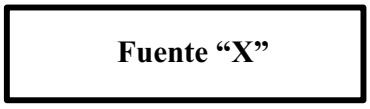


Fuente: Kennet E. Kendal y Julie E. Kendall. **Análisis y diseño de sistemas**. Página 231.

2.2.1.4 Fuente de datos

Las fuentes de datos son personas u organizaciones exteriores al sistema que se analizan, pero que son considerados como emisores o receptores de datos del sistema. Estos son incluidos para añadir comprensibilidad al diagrama y al análisis en general. Son representados mediante cajas con su nombre en el interior.

Figura 11. Representación de una fuente de datos



Fuente: Kennet E. Kendal y Julie E. Kendall. **Análisis y diseño de sistemas**. Página 231.

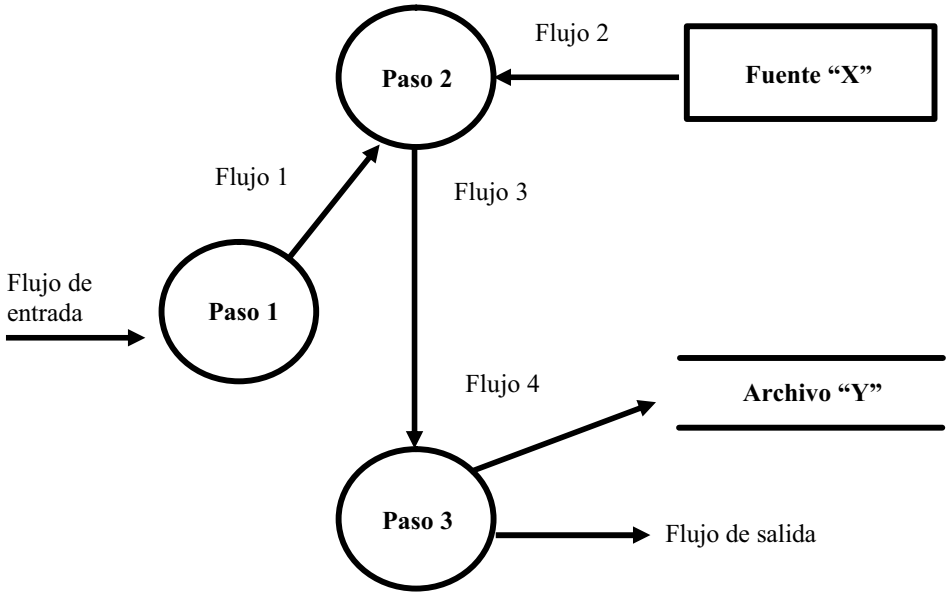
Como la elaboración de un DFD es un proceso que puede ser de tamaño considerable, que consta de varias iteraciones y es realizado de una mejor manera cuando se tiene algo de experiencia, es en realidad algo muy simple. Los pasos básicos para dibujar uno son los siguientes:

1. Identificar las entradas y salidas netas de datos al sistema, para luego dibujarlos en la periferia del diagrama.
2. Ir trabajando desde las entradas hacia las salidas, como lo harían los datos en un recorrido natural o, por el contrario, de atrás hacia adelante. El objetivo es tener un orden determinado que nos ayude a guiarnos.
3. Etiquetar todos los flujos de datos correcta y cuidadosamente. Es decir, con nombres significativos.
4. Etiquetar las burbujas o procesos de acuerdo a sus entradas y salidas.
5. Ignorar las inicializaciones y finalizaciones.
6. Omitir detalles de errores triviales de rutas, al menos en el DFD inicial.
7. No mostrar el flujo de control o información de control.

8. Prepararse para comenzar de nuevo, es decir llevar el DFD inicial al siguiente nivel.

Los DFD pueden componerse por distintos niveles, los cuales detallan los procesos que se llevan a cabo durante cada proceso. Estos niveles aumentan la información y nuestro conocimiento del sistema en su totalidad. Cada uno de los procesos de un DFD se puede descomponer en un DFD interno con sus propios procesos y flujos de datos. La siguiente figura nos ilustra lo anterior.

Figura 12. Ejemplo de un DFD



Fuente: Kennet E. Kendal y Julie E. Kendall. **Análisis y diseño de sistemas**. Página 231.

Como se debe de conservar la coherencia en las relaciones entre los DFD de los distintos niveles, el principal requisito para que este bien desarrollado un DFD en un nivel inferior es que sus entradas de datos deben de ser completamente equivalentes a las entradas de datos que tenga el proceso que lo contiene en el nivel superior inmediato, regla que debe aplicarse también con los flujos de datos de salida.

Finalmente, los procesos dejarán de descomponerse en más DFD cuando se dé una de las siguientes condiciones:

a) Los procesos sean lo suficientemente sencillos como para que su funcionamiento sea obvio y lógico para cualquier persona.
b) Que la complejidad del sistema limite el número de DFD a desarrollar.

En cualquiera de los dos casos, el DFD debe ser lo suficientemente completo para dar una buena explicación del funcionamiento del sistema, sin ser tan vasto y complejo que sea extremadamente difícil su comprensión.

2.2.2 Diccionario de datos

Todos sabemos lo que es un diccionario y para qué se utiliza. Pero, un Diccionario de Datos no es lo mismo, porque contiene definiciones de datos de los elementos de los DFD. Esto incluye las definiciones de los flujos y sus componentes, los procesos y los archivos, así como cualquier otra cosa de la que se necesite una definición.

Y la principal tarea de este tipo de diccionario es darle una verdadera utilidad al DFD, ya que sin estas explicaciones exactas y detalladas de sus componentes, los datos del diagrama podrían ser interpretados de manera distinta por cada persona que lo leyera.

2.2.2.1 Componentes y reglas del diccionario de datos

La parte central del diccionario es la definición. Una definición en un Diccionario de Datos está compuesta por el nombre del elemento a describir (un flujo, un proceso, etc.) y los elementos que construyen a la definición en sí. Por ejemplo:

Datos_de_Cliente = Nombre + Edad + Dirección + Correo_Electrónico
Nombre = Apellido_Paterno + Apellido_Materno + Nombre_de_Pila

Este tipo de definiciones nos proporciona una definición completa y sin lugar a ambigüedad, ofreciendo toda la información que se pueda necesitar para comprender el significado del elemento en cualquier parte del análisis. Pero es obvio que no se puede estar definiendo infinitamente todos los términos utilizados en cada una de las definiciones.

Se debe tener un límite lógico, el cual se pone cuando la persona que realiza el análisis determina que cualquier persona que vaya a leer el Diccionario de Datos pueda comprender un término sin que pueda dejar alguna duda. A este tipo de término lo llamaremos autodefinido. Por ejemplo, en Datos_de_Cliente, cualquier persona, involucrada o no en el análisis, puede comprender el significado de edad, aunque los otros campos puedan requerir un poco más de especificación, como en el ejemplo que se dio de nombre.

Como se puede notar, los términos en una definición se encuentran unidos, y separados al mismo tiempo, por un signo "=", y algunos "+"; éstos representan a los operadores lógicos "IGUAL A" e "Y". De esta manera podemos ver que la definición en si está formada por la unión de varios términos.

Sin embargo, hay ocasiones en que se pueden requerir el uso de otros operadores lógicos, dependiendo de las necesidades del elemento a definir en el sistema que se analiza. Los operadores lógicos que se utilizan en el Diccionario de Datos son:

Tabla V. Operadores lógicos del diccionario de datos

SÍMBOLO	SIGNIFICADO
=	Igual a
+	Y
[ ]	O
|	Separadores entre opciones de O
{ }	Repetición
( )	Opcional

Aparte de estos operadores, también se utilizan los asteriscos "*", para poner comentarios en una definición. A continuación se presentan algunos ejemplos del uso de los operadores.

Transporte = [Avión | Autobús | Automóvil | Barco]
Recibo = Clave_de_Recibo + Fecha + { Detalle } + Total
Detalle = Clave_Pieza + Cantidad + Precio_Unidad + (Descripción) + Subtotal

Ahora veamos el ejemplo siguiente:

Recibo = Clave + Fecha + { Detalle } + Total
Detalle = Clave + Cantidad + Precio_Unidad + Descripción + Subtotal

Es lógico que los recibos utilicen un tipo de clave completamente distinto al que se utilizaría para designar las piezas o unidades que maneja una empresa, y podría considerarse que la diferencia entre ellas es obvia a cualquier persona que las vea. Pero

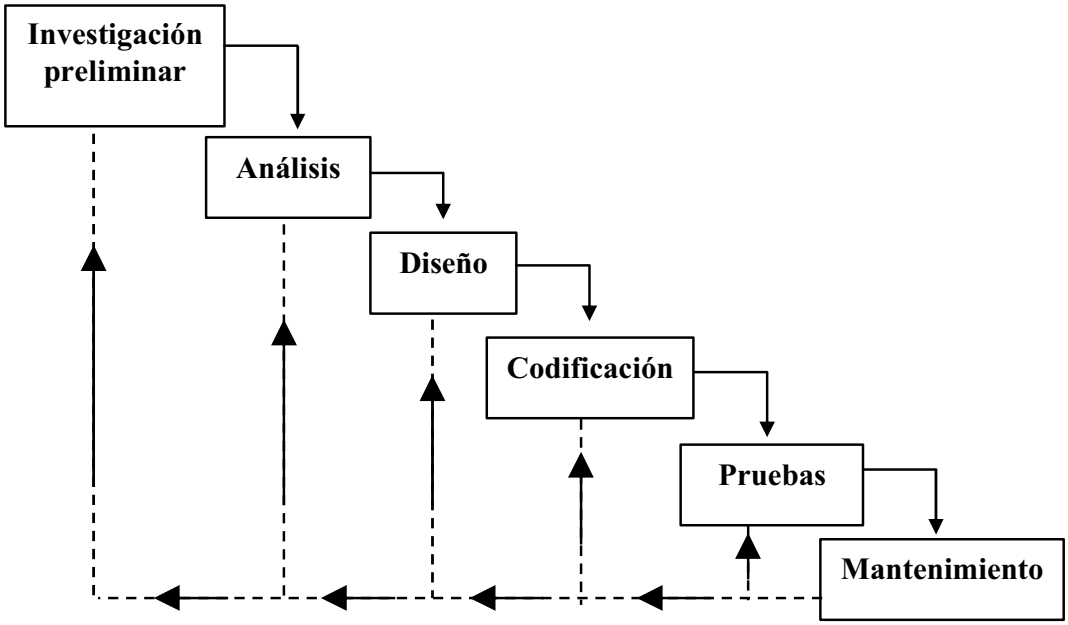
no es así, es incorrecto utilizar un mismo término en el Diccionario de Datos para designar dos elementos distintos y se debe de especificar bien cada uno de ellos.

No solamente se debe evitar utilizar el mismo nombre para dos términos distintos, sino que se debe cuidar el no poner información redundante en el Diccionario de Datos. Además, se debe cuidar que no se hayan asignado dos nombres distintos, por muy parecidos que sean, para un mismo elemento, pues esto resultaría en dos definiciones distintas de la misma cosa. Otro caso en el que se debe cuidar la redundancia de la información es al poner comentarios. No debe estar presente en los comentarios información que puede ser obtenida en otra parte del Diccionario de Datos, o a través de los DFD.

2.2.3 Esquema del modelo estructurado

En la siguiente figura, se muestra el esquema del modelo estructurado, así como sus variaciones con respecto al modelo clásico o en cascada.

Figura 13. Esquema del modelo estructurado



Fuente: **Ciclo vida *software***. http://mural.uv.es/givaro/modulo/Ciclo.htm. (20/08/2002)

2.3 Similitudes entre ambas metodologías

Durante el desarrollo del modelo en cascada, los desarrolladores de *software* se dieron cuenta de la necesidad de retroceder a etapas anteriores del modelo debido a errores cometidos en etapas previas, por ejemplo, una mala definición de requerimientos, un diseño inapropiado, etc. Es por eso que surge la modificación del modelo en cascada, con la innovación de permitir regresar a cualquier etapa previa en

cualquier momento, esto con el fin de corregir los errores cometidos en etapas anteriores antes de que afecten de gran manera al proyecto, por ejemplo atrasos en planificaciones, planificaciones ajustadas como consecuencia de dichos errores, etc.

Este modelo es conocido también con el nombre de modelo en cascada con retroalimentación o modelo en cascada con *feedback*.

3. MODELO DE PROTOTIPOS Y MODELO EN ESPIRAL

3.1 Modelo de prototipos

Un prototipo es un modelo preliminar de la representación, o demostración, fácilmente ampliable y modificable de un sistema previamente planificado, incluyendo interfaz y funcionalidad de entradas y salidas.

El proceso iterativo de construcción de prototipos debe estar precedido por una fase de análisis y construcción del modelo conceptual. Posteriormente, cada una de las iteraciones deberá contar con una fase rápida de análisis y con la confrontación permanente de las necesidades del usuario para que cada iteración nos lleve hacia el sistema final ideal.

3.1.1 Tipos de prototipos

La aplicación de una técnica de prototipos va a ser fundamental en el desarrollo e implementación de los métodos para la inspección y verificación de un producto, debido a que, no será el producto final el que se someta a las diversas pruebas, sino un prototipo del mismo con determinadas características. A continuación se enumeran las diferentes técnicas utilizadas para la implementación de dichos prototipos.

3.1.1.1 Según la fidelidad de la reproducción de la interfaz

3.1.1.1.1 Prototipo de baja fidelidad

Baja fidelidad significa que los prototipos a utilizar no tienen el aspecto real de la interfaz que se está probando, aun cuando operan de la misma forma. La idea es conseguir una gran cantidad de información de la interacción entre la interfaz y el usuario mediante la evaluación de este prototipo. Dado que los prototipos de baja fidelidad son baratos, en términos de dinero, es posible utilizar un mayor número de pruebas, o más prototipos hasta llegar al sistema final.

Esta técnica se utiliza cuando no se dispone todavía de la interfaz real, previsiblemente en las primeras etapas del proceso de desarrollo. Esta técnica es ideal cuando se dispone de poco tiempo y dinero para gastar y resulta de mayor interés la información aportada por el usuario que la reunión de datos.

3.1.1.1.2 Prototipo de alta fidelidad

El prototipo de alta fidelidad es un método donde el prototipo utilizado para las pruebas corresponde con la interfaz real en la mayor medida posible. Normalmente, y en particular para interfaces de *software*, es otra herramienta de *software* la utilizada para diseñar la interfaz. Dicha herramienta acepta entradas desde ratón o teclado, tal y como haría la interfaz real, y responde a esos eventos de idéntica forma, mostrando una ventana en particular, un mensaje, cambiando de estado, etc.

3.1.1.2 Según la funcionalidad reproducida

3.1.1.2.1 Prototipo horizontal

Los prototipos horizontales exhiben muchas de las características del producto final, pero sin el respaldo de una funcionalidad relativamente amplia. Los prototipos horizontales se utilizan con frecuencia para evaluar las preferencias de los usuarios con respecto de las interfaces de usuario. Cuando las funciones reales operativas aún no han sido implementadas, estos prototipos permiten una evaluación del diseño de la interfaz, así como la ubicación y accesibilidad de determinados aspectos y características, sin requerir el funcionamiento real de las funciones que estas representan.

Esta técnica se utiliza de preferencia en las etapas iniciales del proceso de desarrollo, cuando el trabajo sobre las funciones reales del producto aún no ha dado comienzo, pero el conjunto de características es conocido.

3.1.1.2.2 Prototipo vertical

Los prototipos verticales muestran la funcionalidad exacta de un producto para una pequeña parte del conjunto completo. Por ejemplo, un prototipo vertical de un procesador de textos podría mostrar todas las funciones de comprobación de ortografía y gramática, pero ninguna función relacionada con la entrada de texto o su formato. Todas las funciones de un prototipo vertical imitan su equivalente real tanto como sea posible.

Como un prototipo vertical ha de ser funcional prácticamente de forma completa, aunque sólo para una pequeña parte de la interfaz del producto, la mejor forma de obtener un prototipo vertical es utilizar un módulo completamente operativo de un producto. Por ejemplo, para aplicaciones *software* que se han desarrollado con arquitectura modular esto resulta relativamente sencillo, aunque las interfaces a otros

módulos no funcionarán, lo que no supone ningún problema porque es la funcionalidad de las secciones dadas las que serán inspeccionadas o probada, y no otras.

Esta técnica es utilizada cuando el diseño para una parte del producto en particular está prácticamente completo y es conveniente evaluarla en tanto que es un elemento contiguo a otro. Aun cuando algunas partes del producto aún no están listas para la verificación, es posible determinar ciertos problemas con alguna parte en concreto mientras las demás se encuentran aún en fase de desarrollo.

3.1.1.3 Prototipos rápidos

El prototipo rápido se describe como un método basado en pretender reducir las iteraciones en el ciclo de diseño. Habitualmente se desarrollan prototipos que son rápidamente reemplazados o modificados como consecuencia de los datos proporcionados por continuos experimentos. Es un método propio del *software* y la participación del usuario se relega a la verificación del prototipo. Los principales son:

3.1.1.3.1 Desarrollo rápido de aplicaciones (*Rapid Application Development* / RAD)

Es un modelo de proceso del desarrollo del *software* lineal secuencial que enfatiza un ciclo de desarrollo extremadamente corto. El modelo RAD es una adaptación a alta velocidad del modelo lineal secuencial en el que se logra el desarrollo rápido utilizando un enfoque de construcción basado en componentes. El RAD comprende las siguientes etapas:

- ✓ **Modelado de gestión:** aquí se modela el flujo de información entre las funciones de gestión. Este flujo debe responder a preguntas tales como:

- ¿Qué información conduce el proceso de gestión?
- ¿Quién la genera?
- ¿Adónde va la información?
- ¿Quién la procesa?

- ✓ **Modelado de datos:** se definen las características o atributos de cada objeto, formado a partir del flujo de información y las relaciones entre ellos.
- ✓ **Modelado del proceso:** las descripciones del proceso se crean para añadir, modificar, suprimir o recuperar un objeto de datos.
- ✓ **Generación de aplicaciones:** en lugar de crear *software*, el RAD reutiliza componentes de programas ya existentes o crea componentes reutilizables.
- ✓ **Prueba y entrega:** debido al punto anterior, los componentes ya han sido examinados y probados, lo cual permite que el tiempo de duración de las pruebas sea menor. Todo esto no impide que se tenga que probar cada uno de los nuevos componentes.

La metodología RAD plantea cinco elementos claves:

1. Un enfoque de desarrollo iterativo,
2. Fuerte énfasis en el modelado empresarial y de datos,
3. Uso de herramientas automáticas para creación rápida de prototipos y generación de código,
4. Participación activa del usuario final durante todo el ciclo de vida del *software*,
5. Apoyo en código, formas, módulos y objetos reutilizables.

3.1.1.3.2 Desarrollo conjunto de aplicaciones (*Join Application Development* / JAD)

Las sesiones JAD tienen como objetivo reducir el tiempo de desarrollo de un sistema manteniendo la calidad del mismo. Para ello se involucra a los usuarios a lo largo de todo el desarrollo del sistema, es decir, desde la identificación de la necesidad, la propuesta de alternativas de solución y sobre todo en la especificación de los requisitos que debe cubrir el sistema y en la validación de prototipos.

Las características de una sesión de trabajo tipo JAD se pueden resumir en los siguientes puntos:

- Se establece un equipo de trabajo cuyos componentes y responsabilidades están perfectamente identificados, y su fin es conseguir el consenso entre las necesidades de los usuarios y los servicios del sistema en producción.
- Se llevan a cabo pocas reuniones, de larga duración y muy bien preparadas.
- Durante la propia sesión se elaboran los modelos empleando diagramas fáciles de entender y mantener, directamente sobre herramientas CASE.
- Al finalizar la sesión se obtienen un conjunto de modelos que deberán ser aprobados por los participantes.

Es importante definir claramente el perfil y las responsabilidades de los participantes de una sesión JAD. Se pueden distinguir los siguientes perfiles:

- ✓ **Moderador o líder JAD**: persona con amplios conocimientos de la metodología de trabajo, dinámica de grupos, psicología del comportamiento, así como de los procesos de la organización que es objeto del estudio.
- ✓ **Promotor:** persona que ha impulsado el desarrollo.
- ✓ **Jefe de proyecto:** responsable de la implantación del proyecto de informática.

- ✓ **Especialista en modelización:** responsable de la elaboración de los modelos en el transcurso de la sesión.
- ✓ **Desarrolladores:** aseguran que los modelos son correctos y responden a los requisitos especificados.
- ✓ **Usuarios:** responsables de definir los requisitos del sistema y validarlos.

Para llevar a cabo una sesión JAD, es necesario realizar una serie de actividades antes de su inicio, durante el desarrollo y después de su finalización. Estas actividades se detallan a continuación:

- ✓ **Inicio:** se define el ámbito y la estructura del proyecto, los productos a obtener, se prepara el material necesario para la sesión, se determina el lugar donde se van a llevar a cabo, se seleccionan los participantes y se sugiere una agenda de trabajo.
- ✓ **Desarrollo:** se identifican las salidas del proyecto y se debe conseguir el consenso entre los participantes de modo que se materialice en los modelos.
- ✓ **Finalización:** se valida la información de la sesión y se generan los productos de la metodología de trabajo propuesta. Si fuera necesario se integran los productos de salida.

En las sesiones de trabajo tipo JAD se distinguen dos tipos de productos:

1. Dé preparación donde se incluye, entre otros, la historia y contexto del proyecto, los objetivos y límites, las actividades del entorno del negocio que pueden afectar al éxito del proyecto y los beneficios.
2. Dé resultado de las sesiones de trabajo que se establecen con anterioridad al inicio de las reuniones.

3.2 Metodología de desarrollo

Se determina si el sistema que queremos desarrollar es un buen candidato a utilizar el paradigma de ciclo de vida de construcción de prototipos. Por lo general, cualquier aplicación que presente mucha interacción con el usuario, o que necesite algoritmos que puedan construirse de manera evolutiva, iniciando de lo más general y finalizando en lo más específico es un buen candidato para esta metodología.

Sin embargo, hay que tener en cuenta la complejidad del mismo; por ejemplo, si la aplicación necesita que se desarrolle una gran cantidad de código para poder tener un prototipo que enseñar al usuario, las ventajas de la construcción de prototipos se verán superadas por el esfuerzo de desarrollar un prototipo que al final habrá que desechar o modificar mucho. También hay que tener en cuenta la disposición del cliente para probar un prototipo y sugerir modificaciones de los requisitos iniciales para afinar el prototipo.

Es conveniente construir prototipos para probar la eficiencia de los algoritmos que se van a implementar, o para comprobar el rendimiento de un determinado componente del sistema, por ejemplo, una base de datos o el soporte hardware, en condiciones similares a las que existirán durante la utilización del sistema. Es frecuente que el *software* desarrollado presente un buen rendimiento durante la fase de pruebas realizada por los desarrolladores antes de entregarlo al cliente, pero que sea muy ineficiente, o incluso no viable, a la hora de almacenar o procesar el volumen real de información que él debe manejar.

En estos casos, la construcción de un prototipo de parte del sistema y la realización de pruebas de rendimiento sirven para decidir, antes de empezar la fase de diseño, cuál es el modelo más adecuado de entre la gama disponible para el soporte

hardware o cómo deben hacerse los accesos a la base de datos para obtener buenas respuestas en tiempo cuando la aplicación esté ya en funcionamiento.

En otros casos, el prototipo servirá para modelar y poder mostrar al cliente cómo va a realizarse las distintas entradas y salidas de datos en la aplicación, de forma que éste pueda hacerse una idea de como va a ser el sistema final, pudiendo entonces detectar deficiencias o errores en la especificación aunque el modelo no sea más que un prototipo inicial. Por lo tanto, un prototipo puede tener alguna de las tres formas siguientes:

1. En papel o ejecutable en computadora, que describa la interacción hombre-máquina y los listados del sistema.
2. Que implemente algún o algunas de la funciones requeridas, y que sirva para evaluar el rendimiento de un algoritmo o las necesidades de capacidad de almacenamiento y velocidad de cálculo del sistema final.
3. Un programa que realice en todo o en parte la función deseada pero que tenga características que deban ser mejoradas durante el desarrollo del proyecto. Por ejemplo, rendimiento, consideración de casos particulares, etc.

Si el problema aplica para utilizar esta metodología, se debe realizar un modelo del sistema, a partir de los requisitos que ya conozcamos. En este caso no es necesario realizar una definición completa de los requisitos, pero sí es conveniente determinar al menos las áreas donde será necesaria una definición posterior más detallada.

Posteriormente se diseña el prototipo inicial. Se tratará de un diseño rápido, centrado sobre todo en la arquitectura del sistema y la definición de la estructura de las interfaces más que en aspectos de procedimientos de las distintas rutinas, es decir que nos fijaremos más en la forma y en la apariencia que en el contenido.

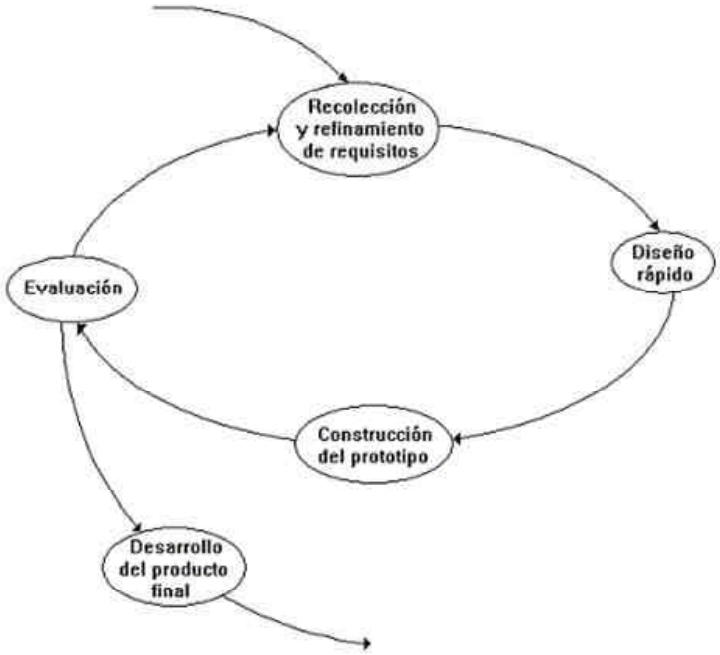
A continuación se construye el prototipo a partir del diseño obtenido. Esta construcción la llevaremos a cabo utilizando herramientas especializadas en generar prototipos ejecutables a partir del diseño, o utilizando técnicas de cuarta generación. Cualquiera que sea la opción elegida, el objetivo es siempre que la codificación sea rápida, aunque la calidad del *software* generado, no sea la deseada.

Una vez finalizado el prototipo inicial, debe ser presentado al cliente para que lo pruebe y sugiera modificaciones. En este punto, el cliente puede ver una implementación de los requisitos que ha definido inicialmente y sugerir las modificaciones necesarias en las especificaciones para que satisfagan mejor sus necesidades.

A partir de estas observaciones hechas por el cliente y los cambios que se muestren necesarios en los requisitos, se procederá a construir un nuevo prototipo y así sucesivamente hasta que los requisitos queden totalmente formalizados, y se pueda considerar el prototipo realizado como el producto final.

3.2.1 Esquema del modelo de prototipos

Figura 14. Esquema del modelo de prototipos



Fuente: **El ciclo de vida**. http://www.lafacu.com/apuntes/informatica/inge_soft/isw2/default.htm. (20/08/2002)

3.3 Modelo en espiral

El modelo en espiral combina las principales ventajas del modelo de ciclo de vida en cascada y del modelo de construcción de prototipos. Proporciona un modelo evolutivo para el desarrollo de sistemas de *software*, es mucho más realista que el ciclo de vida clásico, y permite la utilización de prototipos en cualquier etapa de la evolución del proyecto. Agrega, además, el análisis de riesgo, elemento que en los anteriores modelos estaba ausente. El modelo es representado mediante una espiral, en la cual se definen cuatro actividades principales, correspondiendo cada una a un cuadrante del plano cartesiano.

Las cuatro actividades del presente modelo son las siguientes:

3.3.1 Planificación

Consiste en determinar los objetivos del proyecto, las posibles alternativas y las restricciones. Esta fase equivale a la de recolección de requisitos del ciclo de vida clásico e incluye además la planificación de las actividades a realizar en cada iteración.

3.3.2 Análisis de riesgos

Una de las actividades de la planificación de proyectos es el análisis de riesgos. El desarrollo de cualquier proyecto lleva implícito una serie de riesgos: unos relativos al propio proyecto que son los que pueden hacer que el proyecto fracase, y otros relativos a las decisiones que deben tomarse durante su desarrollo que están asociados a que una de estas decisiones sea errónea. El análisis de riesgos consiste en cuatro actividades principales:

3.3.2.1 Identificar los riesgos

Los riesgos pueden estar presentes en distintos aspectos, por ejemplo en el proyecto tales como presupuestarios, de organización, del cliente. etc., riesgos técnicos tales como problemas de diseño, codificación, mantenimiento, etc.

3.3.2.2 Estimación de riesgos

Consiste en evaluar para cada riesgo identificado, la probabilidad de que éste ocurra y las consecuencias, es decir, el costo que tendrá en caso de que ocurra.

3.3.2.3 Evaluación de riesgos

Consiste en establecer unos niveles de referencia para el incremento de costo, de duración del proyecto y para la degradación de la calidad que, si se superan, harán que se interrumpa el proyecto. Luego hay que relacionar cuantitativamente cada uno de los riesgos con estos niveles de referencia, de forma que en cualquier momento del proyecto podamos calcular si hemos superado alguno de los niveles de referencia.

3.3.2.4 Gestión de riesgos

Consiste en supervisar el desarrollo del proyecto, de forma que se detecten los riesgos tan pronto como aparezcan, se intenten minimizar sus daños y exista un apoyo previsto para las tareas críticas, es decir, aquéllas que más riesgo encierran.

3.3.3 Ingeniería

Consiste en el proceso de codificación del diseño, es decir, la implementación del prototipo en la primera iteración. En las iteraciones siguientes, la fase de ingeniería consiste en codificación de los nuevos requerimientos y mantenimientos para perfeccionar el prototipo, mantenimientos de tipo correctivo para adaptarlo a las necesidades del cliente.

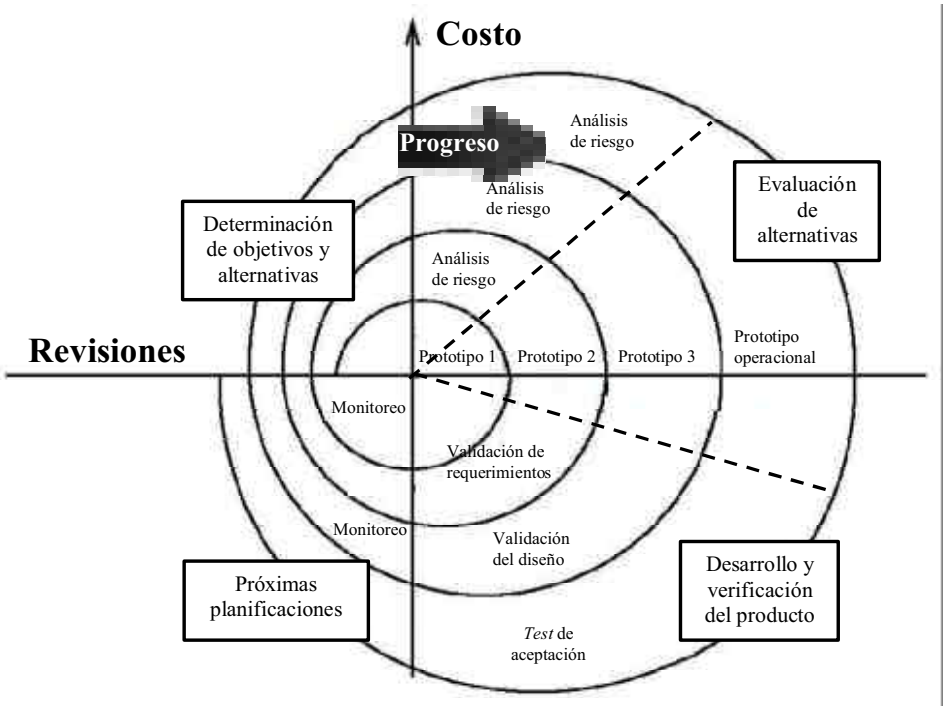
3.3.4 Evaluación del cliente

Consiste en la valoración por parte del cliente de los resultados de la ingeniería. En la primera iteración se definen los requisitos del sistema y se realiza la planificación inicial del mismo. A continuación se analizan los riesgos del proyecto, basándonos en los requisitos iniciales y se procede a construir un prototipo inicial del sistema. Seguidamente, el cliente procede a evaluar el prototipo y con sus comentarios, aprobaciones y desaprobaciones, se procede a refinar los requisitos y a reajustar la planificación inicial, volviendo a empezar el ciclo anteriormente descrito.

En cada una de las iteraciones se realiza el análisis de riesgos, teniendo en cuenta los requisitos y la reacción del cliente ante el último prototipo. Si los riesgos son demasiado grandes se terminará el proyecto, aunque lo normal es que se siga avanzando a lo largo de la espiral. Con cada iteración, se construyen versiones sucesivas del *software*, cada vez más completas, y aumenta la duración de las operaciones del cuadrante de ingeniería, obteniéndose al final el sistema completo.

3.3.5 Esquema del modelo en espiral

Figura 15. Esquema del modelo en espiral



Fuente: **Ciclo de vida del *software***. http://www.geocities.com/Athens/olympus/8740/Ciclo.htm. (10/07/2002)

4. MODELO ITERATIVO INCREMENTAL

El modelo incremental combina elementos del modelo lineal secuencial con la filosofía interactiva de construcción de prototipos. Cuando se utiliza un modelo incremental, el primer incremento a menudo es un producto esencial. Es decir, se afrontan requisitos básicos, pero muchas funciones suplementarias quedan sin extraer.

El modelo afronta la modificación del producto central a fin de cumplir mejor las necesidades del cliente y la entrega de funciones, y características adicionales conforme avanzan las iteraciones del modelo hasta llegar al producto final.

Una iteración a través de las cuatro fases es un ciclo de desarrollo; cada iteración da como resultado una nueva versión del *software*. A menos que el producto no satisfaga las necesidades para las que fue creado, evolucionará en su próxima iteración repitiendo la misma sucesión desde un principio: concepción, elaboración, construcción y transición, pero con un énfasis diferente en las distintas iteraciones.

4.1 Fase de concepción

Es la fase inicial del ciclo de vida. Es quizás la más importante del modelo, porque en ésta se deben identificar todas las necesidades del proyecto, las cuales serán proporcionadas por todos los usuarios que de una u otra forma harán uso de la aplicación, desde los altos ejecutivos hasta los usuarios finales. Son sistemas externos que podrían interactuar con nuestro sistema; lo anteriormente descrito se denomina por convención actor. En otras palabras, un actor es todo aquello que interactúa de forma directa con nuestro sistema.

Los principales objetivos de esta fase son los siguientes:

- Establecer una visión operacional del proyecto de *software* identificando claramente los límites y alcances.
- Identificar claramente los principales casos de uso del sistema.
- Identificar por lo menos una arquitectura para nuestro sistema.
- Realizar una estimación global del proyecto, las cuales se detallaran en la siguiente fase que corresponde a la de elaboración.
- Realizar un análisis de riesgos potenciales, sin faltar los impredecibles.

Para lograr alcanzar los objetivos anteriormente descritos, es necesario llevar a cabo una serie de actividades esenciales que a continuación se enumeran:

- **Alcance del proyecto:** esto involucra capturar el contexto y los requisitos más importantes que a nuestro criterio, como analistas de sistemas consideremos críticos para el correcto funcionamiento del sistema.
- **Arquitectura inicial:** el objetivo aquí es demostrar viabilidad a través de algún tipo de prueba de concepto. Esto puede tomar la forma de un modelo que simula lo que se requiere, o un prototipo inicial que explora lo que se considera que son las áreas de riesgo alto. El esfuerzo del prototipo durante el principio debe limitarse a ganar confianza que una posible solución. La solución final se desarrolla durante la fase de elaboración y construcción.
- **Se prepara el ambiente para el proyecto:** consiste en llevar a cabo una evaluación del proyecto, y de la organización de nuestro equipo de desarrollo, los organigramas genéricos son los siguientes descentralizado democrático, descentralizado controlado y centralizado controlado. Además, se debe seleccionar las herramientas de desarrollo.

Lo anterior queda plasmado en un documento que generalmente se denomina ***documento de visión***, su análogo con la metodología tradicional o estructurado moderno es el DERCAS (Documento de Especificación de Requerimientos y Criterios de Aceptación del *Software*).

4.2 Fase de elaboración

La meta de la fase de elaboración es la arquitectura del sistema para mantener una base estable y poder aplicar un plan de trabajo en la fase de la construcción. Para alcanzar esta meta es necesario cumplir, como mínimo, con los siguientes objetivos:

- Asegurar que la arquitectura, requisitos, planes son lo suficientemente estables, y los riesgos han sido minimizados para ser capaz de determinar el costo y llevar a cabo la fase de construcción o desarrollo.
- Producir un prototipo evolutivo de calidad, así como uno o los prototipos específicos para mitigar riesgos específicos como:
    - Errores en diseño conceptual.
    - Errores en la determinación de requerimientos.
    - Viabilidad del producto para demostrar a los inversionistas, clientes, y usuarios finales la funcionalidad del producto.
- Se debe demostrar que la arquitectura del sistema apoyará los requisitos del mismo a un costo razonable y en un tiempo justificable.

Para lograr estos objetivos es necesario llevar a cabo las siguientes actividades:

- Definiendo y validando la arquitectura de una manera practica y en el menor tiempo posible.

- Redefinir el documento de visión, actualizándolo con la nueva información obtenida durante la fase y establecer un entendimiento sólido de los casos de los casos de uso más críticos que manejan la funcionalidad del sistema.
- Creando las normas y detallar los planes de la iteración que se llevaran a cabo en la fase de la construcción.
- Se debe refinar la arquitectura del sistema y seleccionar los componentes necesarios. Se evalúan componentes potenciales para su desarrollo si se trata de *software* y adquisición si es hardware. Los componentes arquitectónicos seleccionados se integran y se evalúan contra la arquitectura inicial.

4.3 Fase de construcción

La meta de la fase de la construcción está en clarificar los requisitos restantes y completar el desarrollo del sistema basándose en la arquitectura seleccionada. En la fase de construcción se pone énfasis en los distintos recursos tanto humanos como no humanos para perfeccionar la calidad del producto.

En esta fase se deben de cumplir los siguientes objetivos:

- Minimizar el costo de desarrollo para optimizar recursos y evitar trabajo innecesario debido a una mala planificación.
- Se debe desarrollar versiones útiles y practicas en el tiempo estimado.
- Describir los casos de uso restantes y otros requisitos, para afinar la aplicación, y efectuar pruebas al *software*.
- Se pueden efectuar actividades en paralelo con los equipos de desarrollo de proyectos más pequeños, componentes que pueden progresar independientemente entre sí sin alterar la naturaleza del modelo. Este paralelismo puede acelerar las actividades de desarrollo significativamente, pero también

aumenta la complejidad al dirigir y asignar los recursos y sincronizando el flujo de trabajo.

4.4 Fase de transición

El enfoque de la fase de transición es asegurar que el *software* está disponible para todos los usuarios. Esta fase permite medir por medio de varias iteraciones la calidad del producto, probando el producto y haciendo ajustes menores basados en las necesidades de los usuarios. A estas alturas del ciclo de vida, las necesidades del usuario deben ser principalmente en afinación del producto, configuración e instalación. Y todos los problemas estructurales debieron de haber sido identificados en las etapas previas a la transición.

Las actividades realizadas durante una iteración en la fase de transición dependen de la complejidad de la meta planteada. Por ejemplo, un prototipo con una funcionalidad alta requerirá más actividades que un prototipo funcional que posteriormente será refinado por medio de los requisitos identificados por los usuarios. Sin embargo, esta fase debe de cumplir con los siguientes objetivos:

- Se debe probar y validar el nuevo sistema contra las expectativas de los usuarios.
- Entrenamiento y capacitación de usuarios.
- Se debe contar con documentación y / o material de apoyo para los usuarios.
- Conseguir retroalimentación de los usuarios que será utilizada para planificar la siguiente iteración.
- Se debe desarrollar un producto disponible para todos los usuarios.

Cada una de estas etapas posee un elemento que es llamado iteración. Cada una es un ciclo de desarrollo del proyecto en donde se fijan características. La iteración posee un tiempo definido y abarca un grupo de funciones específicas. Al final de cada

etapa, la idea es tener un prototipo ya armado y funcionando, que es un *software* más pequeño que el deseado, pero que es completamente operativo.

A través de cada etapa del desarrollo, las iteraciones se van distribuyendo según la cantidad de requerimientos. Pero, en general, se puede tener un número fijo de iteraciones por etapa:

a) **Concepción:** 1 iteración que básicamente solo es la definición del alcance del proyecto y su planificación.
b) **Elaboración:** 1 a 3 iteraciones en donde se reúnen los requerimientos más detallados, se realiza el análisis y diseño de alto nivel para definir la arquitectura base, y se crea el plan de construcción.
c) **Construcción:** muchas iteraciones dividiendo la cantidad de requerimientos funcionales a través del tiempo para construir los requerimientos en productos o prototipos incrementales.
d) **Transición:** 1 iteración que básicamente es la implantación y puesta en marcha del proyecto. Se incluye la capacitación y la afinación del desempeño, aunque esto puede ser otra iteración adicional.

Es importante mencionar que para el desarrollo con este proceso, la metodología utiliza otros elementos que definen cómo trabajar, a esto se le llama roles. Los roles dentro del proceso de desarrollo cumplen la función de indicar quién debe ser el responsable de cada uno de los pasos en una iteración.

4.5 Introducción al lenguaje unificado de modelado (UML)

El UML es una metodología orientada a objetos y fue desarrollada por Grany Booch, James Rumbaugh e Ivar Jacobson, en los años ochenta y principios de los noventa. Esta metodología está compuesta por una serie de elementos gráficos que combinados entre sí forman lo que se conoce como diagrama. Permiten al analista de sistemas generar un anteproyecto que cuenta con varias vistas o perspectivas del sistema que sean comprendidas de manera fácil para los clientes, desarrolladores y todos aquellos que están involucrados en el proceso de desarrollo.

Los distintos diagramas de UML son los siguientes: objetos, casos de uso, estados, secuencias, actividades, colaboración y distribución. Y por ser una metodología orientada a objetos las relaciones existentes son distintas que las utilizadas por los modelos anteriores. A continuación se describen las relaciones existentes para posteriormente explicar cada uno de los diagramas antes mencionados.

4.5.1 Teoría de objetos

Por tratarse de una metodología orientada a objetos, es necesario tener un conocimiento general de la teoría de objetos, comprender lo que es un objeto. Un objeto es una instancia particular de una clase o categoría de objetos, que posee una estructura que esta formada por atributos o propiedades y acciones o métodos. Además de lo anterior, los objetos poseen abstracción, herencia, polimorfismo, encapsulamiento, envío de mensajes. Cada uno de estos términos se refiere a lo siguiente:

4.5.1.1 Conceptos generales

a) **Abstracción:** se refiere a quitar las propiedades y acciones de un objeto para dejar sólo aquellas que sean necesarias.
b) **Polimorfismo:** se da cuando existe un método con el mismo nombre en distintas clases, y cada uno realiza acciones diferentes.
c) **Encapsulamiento:** se refiere a que un objeto puede ocultar su funcionalidad a otros objetos.
d) **Envío de mensajes:** debido a que los objetos trabajan en conjunto, la comunicación entre ellos es por este método.

4.5.1.2 Relaciones

Las relaciones estarán presentes en la mayoría de los diagramas de UML, y como el objetivo de esta sección es centrarnos en la notación de éstas, representaremos una clase en su forma simplificada, pero posteriormente se indicará la representación correcta de las mismas.

4.5.1.2.1 Asociación

Existe una relación asociativa cuando dos o más clases se conectan entre sí de manera conceptual. Una relación asociativa puede ser unidireccional o bidireccional, además, podemos indicar restricciones tales como orden y decisiones por medio del operador lógico OR. Este tipo de relación es conocido también con el nombre de relación O, e indicar la cantidad de objetos de una clase que pueden relacionarse con un objeto de una clase asociada, a esto último se le conoce con el nombre de multiplicidad, y se representa por medio de un número entero no negativa. Las relaciones se dibujan con una línea sólida y por medio de un triángulo relleno se indica su dirección. Si existe un orden se indica por medio de la palabra ordenado entre llaves. Y la relación O por

medio de una línea punteada y la palabra OR entre llaves. Tal como se indica en la siguiente figura.

Figura 16. Relación asociativa

Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 48.

Sin embargo, existen unas abreviaturas en UML para representar los distintos tipos de multiplicidad que pueden surgir en una relación, y se muestran a continuación:

Tabla VI. Tipos de multiplicidad

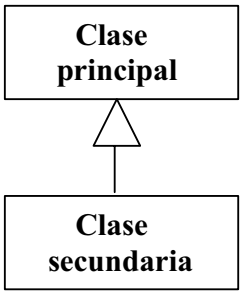
MULTIPLICIDAD	SÍMBOLO
Uno	1
Muchos	*
Uno a más	1..*
Ninguno o uno	0,1

4.5.1.2.2 Herencia

Se conoce también con el nombre de generalización. Se refiere a que una clase posee todas las características de la clase de la que proviene. Para representar este tipo de relación, se unen la clase principal con la clase secundaria por medio de una línea sólida, y en la parte de la línea que conecta a la clase principal se coloca un triángulo sin rellenar que apunte a la clase principal. Este tipo de relación se puede interpretar de la

siguiente forma: la clase secundaria es un tipo de la clase primaria. Esta relación se representa gráficamente como se muestra en la siguiente figura:

Figura 17. Relación herencia o generalización



Fuente: Joseph Schmuller. **Aprendiendo UML en 24** Horas. Página 52.

4.5.1.2.3 Dependencia

Se dice que una clase depende de otra, si una clase utiliza otra. Este tipo de relación se representa por medio de una línea punteada con un triángulo sin relleno en el extremo de la línea que apunta hacia la clase de la que depende. Para su mejor comprensión, observe la siguiente figura:

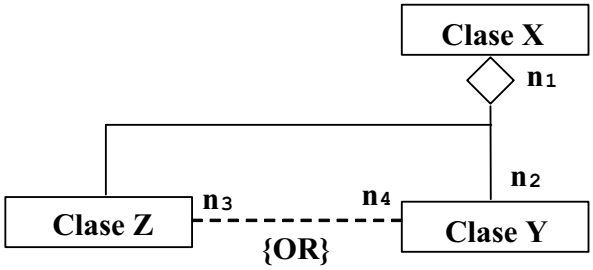
Figura 18. Relación de dependencia

Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 55.

4.5.1.2.4 Agregación

Se refiere a que una clase puede estar formada o compuesta por una serie de clases. La agregación se puede representar como una jerarquía en donde la clase principal se coloca al principio y todas las clases que forman a la clase principal se colocan debajo de ella. Al igual que las relaciones asociativas, se puede indicar que una clase u otra es parte del todo, e indicar multiplicidades. Se representa por medio de una línea sólida que une a las clases y un rombo sin relleno apuntando a la clase principal. Para comprender de una mejor manera, observe la siguiente figura:

Figura 19. Relación de Agregación

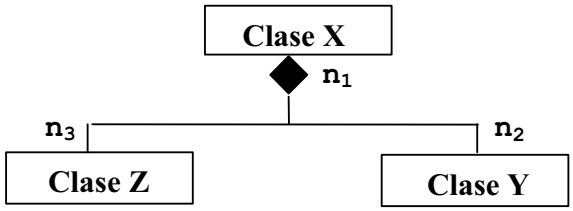


Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 58.

4.5.1.2.5 Composición

Este tipo de relación es similar a la relación de agregación, con la única diferencia de que cada subclase puede pertenecer únicamente a una clase principal. Se representa de la misma manera que una relación de agregación, con la diferencia de que el rombo esta relleno.

Figura 20. Relación de composición



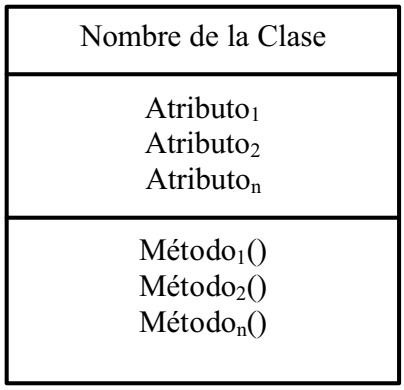
Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 59.

4.5.2 Diagramas de clases

Definiremos una clase como una categoría o grupo cosas tangibles o intangibles que poseen atributos y acciones. Es indispensable para el análisis del sistema, porque cada sustantivo que aparezca en nuestro sistema se transformará en una clase en el sistema, los verbos relacionados al sustantivo serán sus acciones o métodos y lo que se dice del sustantivo serán sus características o atributos. Para representar una clase, lo haremos por medio de un rectángulo dividido en 3 secciones que corresponden a su nombre o identificador, sus atributos y sus métodos respectivamente.

La siguiente figura nos muestra su esquema general:

Figura 21. Representación de una clase

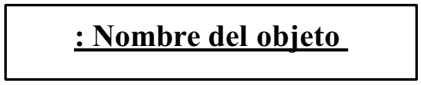


Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 34.

4.5.3 Diagramas de objetos

Definiremos a un objeto como una instancia particular de una clase, por ejemplo, una clase podría ser persona que posee atributos y características generales, y un objeto podría ser usted mismo, que posee esas mismas características pero que son únicas en la clase persona. Para representar una clase lo haremos colocando dos puntos y a continuación el nombre de la instancia u objeto, todo esto subrayado. Para comprender de mejor manera observe la siguiente figura:

Figura 22. Representación de un objeto



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 104.

4.5.4 Diagramas de casos de uso

Los casos de uso le sirven al analista de sistemas para comprender de una mejor manera el funcionamiento del sistema y a representar los requerimientos desde el punto de vista de los usuarios.

En otras palabras, un caso de uso puede ser una función o proceso que realiza el sistema; en un caso de uso, siempre se encuentran involucrados al menos dos actores: el que solicita alguna información y el que la recibe.

Se representa gráficamente de la siguiente forma: una elipse representa el caso de uso, su nombre aparece dentro o debajo de ella, una figura humana representa a los actores, sus nombres correspondientes aparecen debajo de ellos. Una relación de tipo asociativa conecta a los actores con el caso de uso. Un rectángulo con el nombre del sistema ubicado dentro de él, representa los límites del sistema.

4.5.4.1 Relaciones en los casos de uso

Las relaciones, no existen únicamente en las clases, están presentes también en los casos de uso, y pueden ser de los siguientes tipos:

4.5.4.1.1 Inclusión

Esta relación permite nuevamente utilizar el funcionamiento de un caso de uso dentro de otro. Se representa gráficamente de la misma manera que una relación de dependencia, con la diferencia que se etiqueta la relación con la palabra “incluir” dentro de paréntesis angulares.

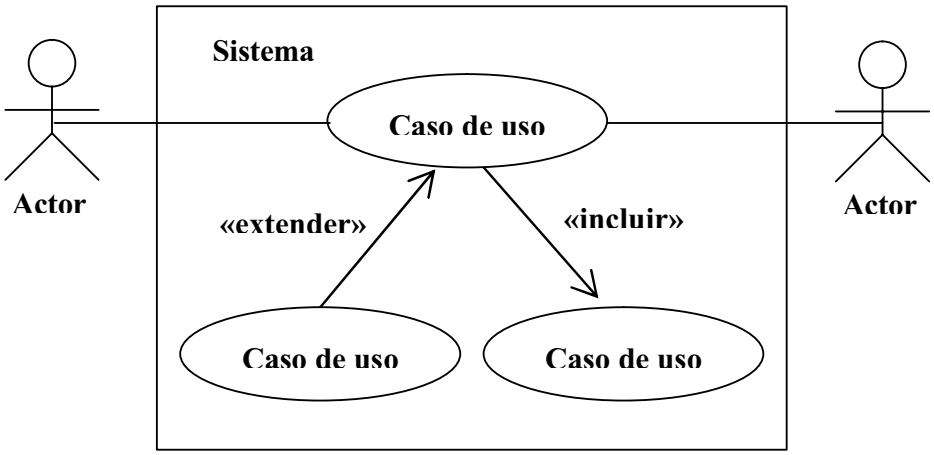
4.5.4.1.2 Extensión

Esta relación permite crear un nuevo caso de uso mediante la utilización de un caso de uso existente, que también se denomina caso de uso base.

Se representa gráficamente de la misma manera que una relación de inclusión, con la diferencia que se etiqueta la relación con la palabra “extender”.

Para comprender de una mejor manera el concepto de caso de uso, observe la siguiente figura en donde se muestra lo explicado anteriormente.

Figura 23. Representación de un caso de uso



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 79.

4.5.5 Diagramas de estados

Estos diagramas se utilizan para representar un cambio en el sistema. Dicho de otra manera, los objetos que forman el sistema modifican su estado como resultado de eventos realizados en el transcurso del tiempo.

Sin embargo, para comprender mejor la definición anterior, es conveniente definir lo que significa estado en los diagramas de UML. Un estado representa un grupo de características definidas de un objeto que pueden cambiar sólo a través de una acción.

Un estado se representa por medio de un rectángulo con vértices redondeados, junto con una línea sólida y una punta de flecha que representan una transición. Esta flecha apunta al estado donde se realizará la transición. Un circulo relleno mostrará el punto de partida de la transición y un círculo relleno sobre uno vacío mostrara el punto de culminación.

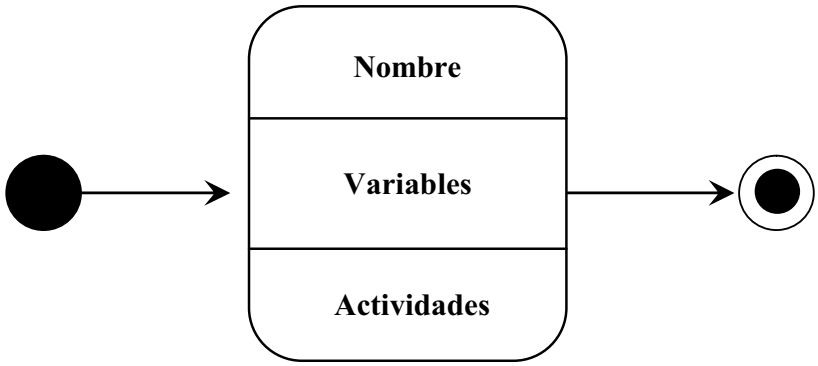
Este rectángulo puede dividirse en 3 áreas en las cuales la primera identificará al estado con un nombre único, la siguiente área corresponde a las variables que nos permiten controlar y saber que se involucra en ese estado y, finalmente, la última área corresponde a las actividades que representan la misión que el objeto cumple en ese estado.

Las actividades se clasifican en 3 tipos que a continuación se describen:

a) Entrada actividades que se realizan al entrar al estado.
b) Se debe hacer actividades realizadas mientras se está en el estado.
c) Salida eventos que se ejecutan al abandonar el estado.

Lo anteriormente expuesto se muestra en la siguiente figura:

Figura 24. Representación de un diagrama de estado



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 93.

4.5.5.1 Subestados

Debido a que un determinado estado puede contener dentro de sí mismo otros estados, estos diagramas son recursivos. Y se pueden definir dos tipos de subestados.

4.5.5.1.1 Subestados secuenciales

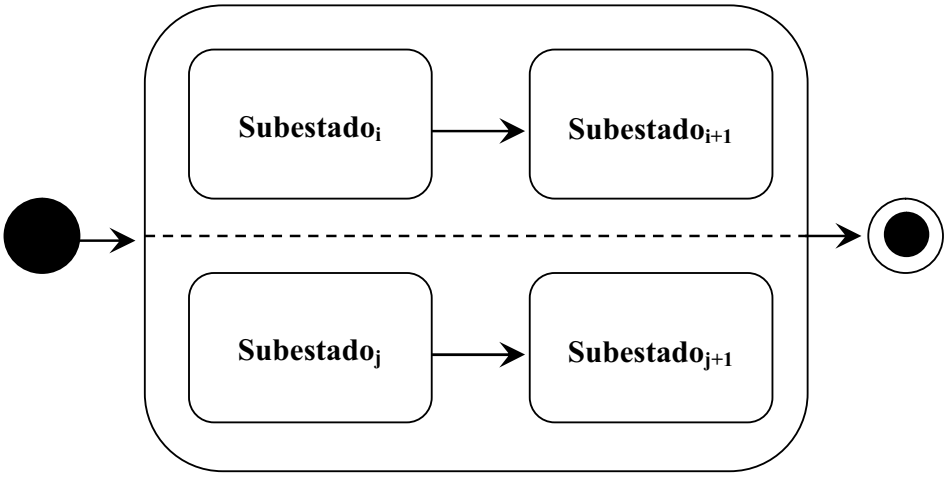
Son aquellos que suceden uno detrás de otro. Es decir, tienen una secuencia u orden claramente definido.

4.5.5.1.2 Subestados concurrentes

Son aquellos que suceden o se ejecutan al mismo tiempo. Para representar la concurrencia gráficamente, se dibuja una línea no continua para separar aquellos subestados secuenciales de los concurrentes.

Lo anterior se explica de mejor manera en la siguiente figura:

Figura 25. Representación de estados y subestados



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 98.

4.5.6 Diagramas de secuencia

Como se ha dicho anteriormente, los objetos se comunican por medio de mensajes, y cambian su estado con el transcurso del tiempo. El diagrama de secuencias sirve para definir la secuencia cronológica de estos eventos.

Y para ello hace uso de los objetos, los cuales se representan por medio de un rectángulo con su nombre subrayado. Mensajes representados por líneas sólidas con una punta de flecha pueden ser simples, es decir que el control de la operación es transferido de un objeto a otro. Sincrónico significa que el objeto que envió el mensaje esperara una respuesta para continuar su ejecución, y asincrónico indica que puede continuar operando sin esperar una respuesta a cambio. Además el tiempo se representa por medio de una línea no continua vertical, denominada línea de vida.

Todos los objetos deben colocarse en la parte superior del diagrama ordenado de izquierda a derecha. Debajo de cada objeto se dibuja una línea de vida. Un pequeño rectángulo denominado activación nos indica la realización de un evento por el objeto, y la longitud del rectángulo marcará su tiempo de duración.

En los sistemas, algunas operaciones son resueltas por medio de llamadas recursivas. Esto en los diagramas de secuencia se representa por medio de una flecha que nace en la activación y se dirige a otro rectángulo colocado en la activación.

4.5.6.1 Diagrama de secuencias de instancias

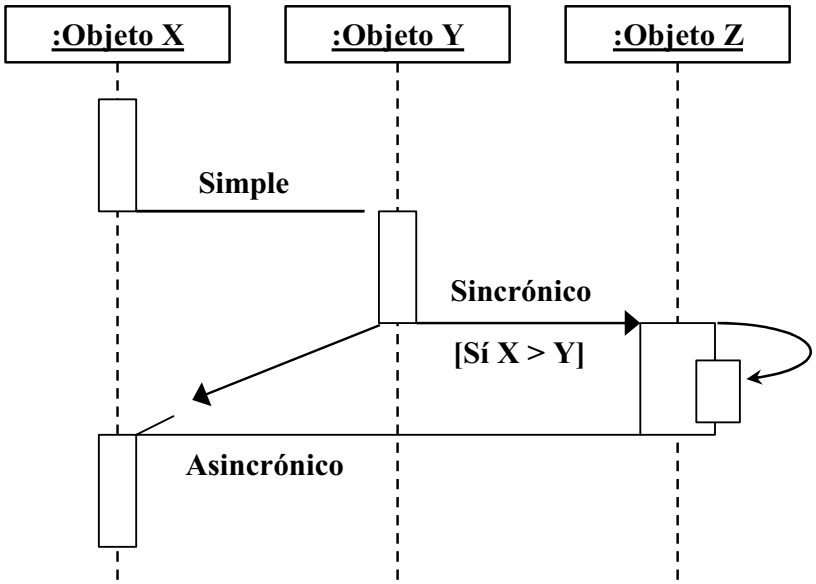
Es todo diagrama en donde el diagrama se centra únicamente en una instancia o escenario. Por lo general, este diagrama muestra una secuencia ideal de los eventos, es decir, todos los eventos suceden sin ningún problema.

4.5.6.2 Diagrama de secuencias genérico

Este diagrama, por el contrario, concentra en uno solo todas las posibles instancias o escenarios de un diagrama específico. En este tipo de diagramas, todo tipo de condiciones se indican entre corchetes.

La siguiente figura muestra la forma general de un diagrama de secuencias:

Figura 26. Representación de un diagrama de secuencias



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 111.

4.5.7 Diagramas de colaboración

Los diagramas de colaboración son similares a los de secuencia. La diferencia radica en que los diagramas de secuencia muestran la sucesión u orden en el que se realizan las interacciones. Los diagramas de colaboración muestran el contexto y organización general de los objetos ínter actuantes. En otras palabras, los diagramas de secuencia se organizan basándose en el tiempo, y los diagramas de colaboración basándose en el espacio.

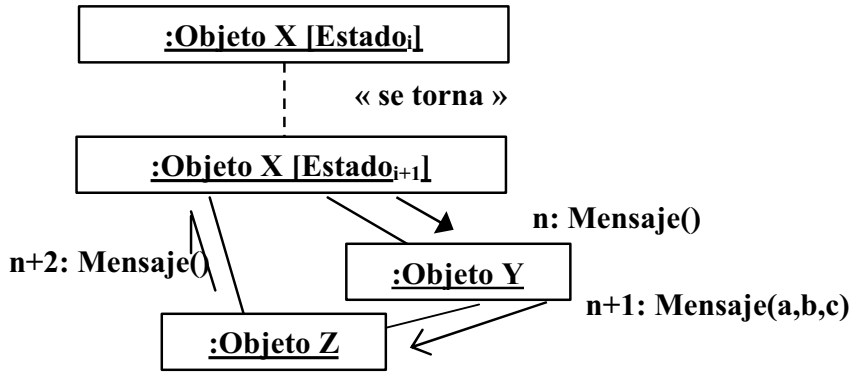
Los objetos que intervienen en el diagrama deben estar de alguna manera asociados, y se representan de la misma manera que en el diagrama de secuencias. Los

mensajes de igual forma, con la diferencia que se finalizan con un par de paréntesis en donde se indican los parámetros del mensaje si fuera necesario.

Además, se puede indicar el estado de un objeto en el diagrama. Esto se hace colocando al lado del nombre del objeto, encerrado entre corchetes su estado. Otro objeto con el mismo nombre y distinto estado estará unido por una línea no continua etiquetada con el estereotipo se torna al estado anterior. Esto se hace con el fin de hacer notar su cambio de estado.

Al inicio se indicó que este diagrama era similar al de secuencia. Para representar la secuencia u orden en el diagrama de colaboración, se indica el orden al inicio de los mensajes que se envían los objetos que forman el diagrama. Lo anterior se comprende de una mejor manera con la siguiente figura:

Figura 27. Representación de un diagrama de colaboración



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 121.

4.5.8 Diagramas de actividades

Estos diagramas muestran de manera simplificada el funcionamiento de una operación o proceso. Los componentes que se utilizan son los mismos que en el diagrama de estados, con la diferencia que aquí un estado representa una actividad, y ésta no posee ni variables ni métodos.

Visto de manera análoga, es similar al diagrama de flujo de datos de otras metodologías. En este diagrama, por lo general es necesario indicar una toma de decisión o elegir entre una y otra actividad. Esta decisión se puede representar de una de dos formas, no se deben de mezclar ambas representaciones en un diagrama.

La primera es dibujar todos los caminos posibles que nacen en una actividad. La segunda forma es dibujar al final de la transición un rombo, del cual deben de salir todos los caminos posibles.

Si en nuestro diagrama es necesario modelar que dos o más caminos se ejecutan al mismo tiempo, se dibuja una línea sólida gruesa al inicio y al final de los caminos involucrados.

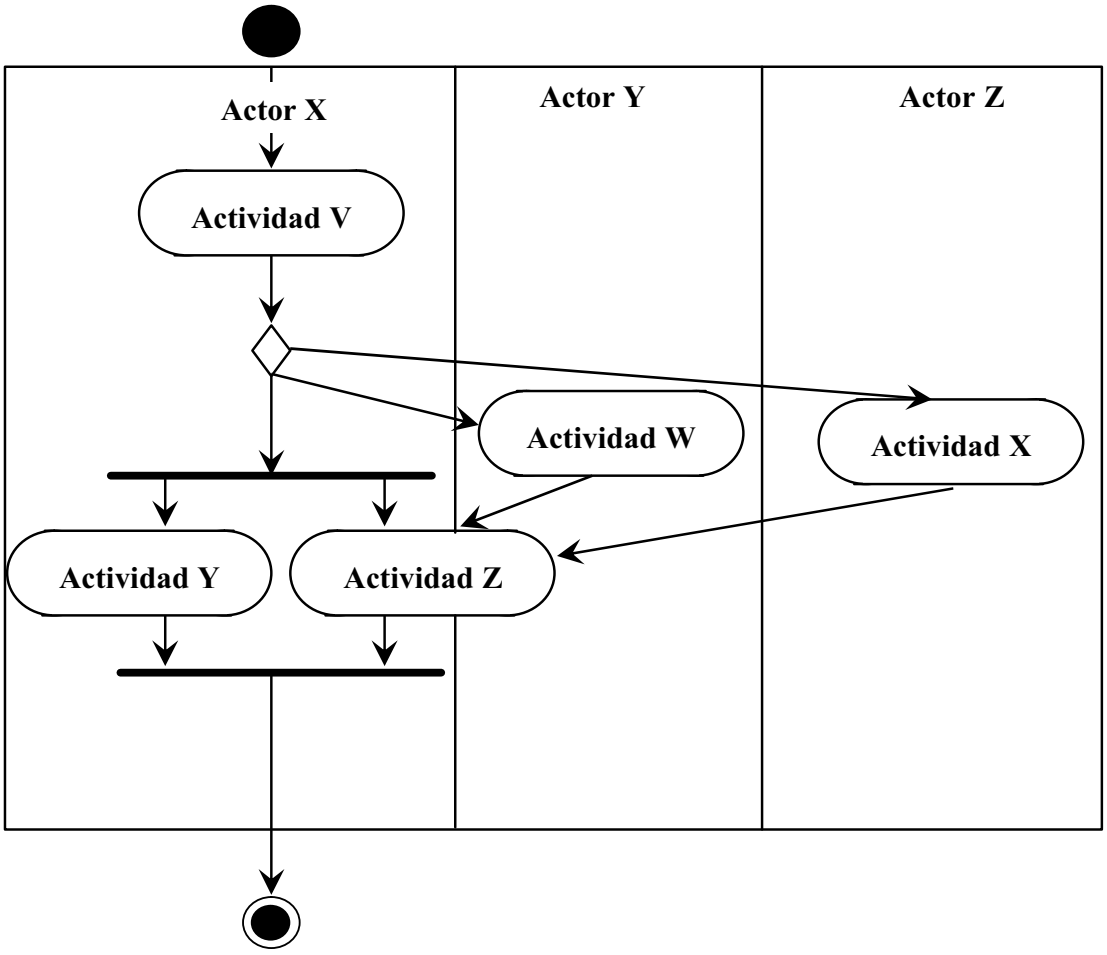
4.5.8.1 Marcos de responsabilidad

El diagrama de actividades nos da la posibilidad de indicar qué actor es el responsable de una actividad específica. Esto le agrega al diseño desde su concepción seguridad al sistema, elemento que en las otras metodologías se implementaba de otra manera.

Para representar esto gráficamente, se debe dividir el diagrama de actividades en segmentos paralelos llamados marcos de responsabilidades. Cada marco debe mostrar el

nombre de un responsable en su parte superior y debe mostrar sus actividades asociadas. Las transiciones entre las distintas actividades pueden darse entre marcos distintos. La siguiente gráfica muestra lo anteriormente expuesto:

Figura 28. Representación de un diagrama de actividades



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 142.

4.5.9 Diagramas de componentes

Este diagrama nos permite representar partes específicas del sistema. Recordemos que un sistema está formado por un conjunto de elementos o componentes que están relacionados entre sí, con un objetivo en común. Para comprender la definición anterior, es indispensable comprender el concepto de componente.

4.5.9.1 Componente

Representa una parte de nuestro sistema, por ejemplo, una base de datos, archivos, librerías, etc., que tiene una función específica. Algunos de los objetivos de modelar el sistema basándose en componentes son por las siguientes razones:

- El cliente puede observar de manera clara y legible la estructura final del sistema.
- Los programadores tendrán una estructura sobre la cual empezar el desarrollo del sistema.
- Las personas encargadas de elaborar toda la documentación del sistema entenderán de mejor manera el funcionamiento del sistema.
- Los componentes pueden ser reutilizados por otro sistema, con lo cual se estará ahorrando tiempo y esfuerzo en modelar un componente nuevo.

Una característica de los componentes es que sus operaciones solamente pueden ser ejecutadas desde una interfaz. Sin embargo, todavía no se ha definido el significado de la misma, y por ella entenderemos lo siguiente:

4.5.9.2 Interfaz

Es el medio por el cual el sistema se comunicará con los usuarios. En otras palabras, es el medio por el cual el sistema nos solicitará información y los usuarios pueden ingresar la misma.

Para organizar los componentes en nuestro sistema, los podemos clasificar de la siguiente manera:

a) **Componentes de distribución:** son la base de los sistemas ejecutables, por ejemplo, librerías de acceso dinámico (DLL *Dynamic Library Linck*), controles *ActiveX*, etc.
b) **Componentes para trabajar en el producto:** a partir de estos componentes se crean los componentes anteriores. Por ejemplo, archivos de base de datos, archivos fuente, etc.
c) **Componentes de ejecución:** se generan como resultado de ejecutar un componente. Por ejemplo, en *Windows* un componente de ejecución son los archivos de ayuda (.HLP). Cuando se genera un índice de ayuda por primera vez, se crea un componente para trabajar con el producto. Este índice es almacenado en un archivo de búsqueda de texto (FTS). Este archivo generado como resultado de las operaciones anteriores, es considerado como un componente de ejecución.

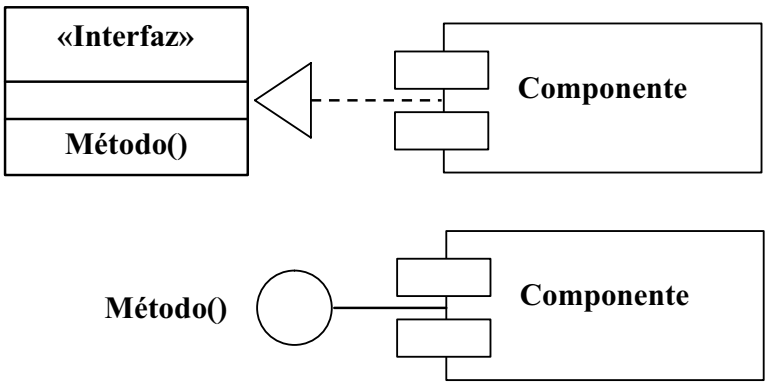
Para dibujar un componente, lo haremos por medio de un rectángulo que posee otros dos rectángulos más pequeños del lado izquierdo. Y las interfaces las podemos representar de dos maneras:

1. La primera con un rectángulo con información relacionada y conectada al componente por una línea no continua, y un triángulo vacío en el extremo que conecta a la interfaz.

2. La segunda, por medio de un círculo conectado por medio de una línea sólida continua.

Para comprender de mejor manera lo anterior, la siguiente figura muestra las alternativas de diseño.

Figura 29. Representación de un diagrama de componentes



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 152.

4.5.10 Diagramas de distribución

El diagrama anteriormente expuesto sirve para modelar el sistema en su parte intangible o *software*. El diagrama de distribución sirve para complementar nuestro modelo, es decir, sirve para modelar la arquitectura de hardware que va a tener nuestro sistema final.

Para modelar la arquitectura del sistema, lo haremos por medio de nodos, que no son más que un elemento de hardware que puede ser de dos tipos:

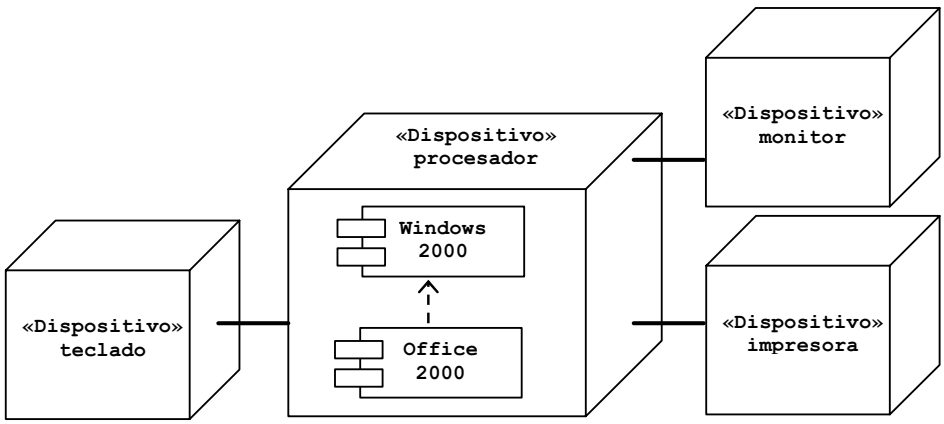
1. **Nodo procesador:** es aquel que ejecuta uno o varios componentes.
2. **Nodo dispositivo:** es aquel que no ejecuta componentes, pero tiene contacto de alguna manera con el mundo exterior. Por ejemplo, una impresora, un monitor, etc.

En la notación de UML, un nodo se representa por medio de un cubo, el cual debe de ir identificado con un nombre y puede estar dividido en secciones para agregar información que se considere necesaria.

Una vez colocados todos los elementos hardware de nuestro sistema, tenemos que relacionarlos para indicar la conexión existente entre ellos. Para hacer esto debemos dibujar una línea sólida entre los nodos que estén relacionados.

Para comprender de una mejor manera este diagrama, observe la siguiente figura, la cual modela una computadora personal.

Figura 30. Representación de un diagrama de distribución



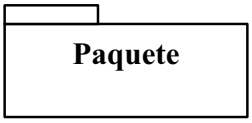
Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 166.

4.5.11 Elementos especiales

4.5.11.1 Paquete

Este elemento sirve para organizar los distintos elementos en un diagrama. Por ejemplo, clases, componentes, etc. Para representar un paquete, se hace una carpeta tabular como en la figura que se muestra a continuación:

Figura 31. Representación de un paquete

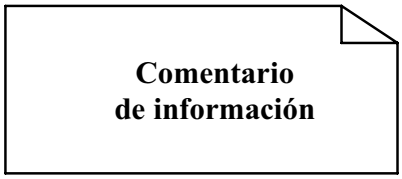


Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 171.

4.5.11.2 Notas

Este elemento es muy útil para documentar los distintos diagramas. En este elemento colocaremos todo tipo de comentarios que consideremos necesarios para una mejor explicación del diagrama. Para hacer una anotación, dibujaremos un rectángulo con la esquina superior derecha doblada, como se muestra en la siguiente figura:

Figura 32. Representación de una nota



Fuente: Joseph Schmuller. **Aprendiendo UML en 24 Horas**. Página 171.

5. OTROS MODELOS DE DESARROLLO

Para desarrollar un producto de *software* no existe una metodología perfecta porque cada proyecto tiene diferentes requerimientos, y cada una posee ventajas y desventajas. Para disminuir de alguna manera estas desventajas, han aparecido nuevas metodologías o técnicas de desarrollo que no son más que variaciones de las metodologías tradicionales.

En el presente capítulo se describen estas metodologías que, por ser variaciones de las metodologías tradicionales, son poco conocidas. Y por esa razón es importante darlas a conocer, para crear una nueva cultura y comenzar a aplicarlas en el desarrollo de *software*.

5.1 Programación extrema (XP)

La programación extrema se basa en una serie de reglas y principios que han surgido a lo largo de toda la historia de la ingeniería del *software*. El principal objetivo de la programación extrema consiste en controlar el problema de las especificaciones incompletas, cambios y falta de cultura del negocio que provoca que los desarrolladores tarden más en entender el sistema que deben desarrollar.

Se ataca este problema mediante ciclos de iteraciones cortos en los cuales se agrega muy poca funcionalidad al sistema, esto permite a los desarrolladores entender la problemática de una forma más natural. Pero exige una alta participación de los usuarios en el desarrollo para crear un auténtico ambiente de trabajo en equipo.

La Programación Extrema define las siguientes cuatro variables en todo proyecto de *software* (costo, tiempo, calidad y alcance), de las cuales las tres primeras son establecidas por fuerzas externas al proyecto, por ejemplo el cliente, los jefes de proyecto, etc. La variable alcance es definida por el equipo de desarrollo en función de las anteriores.

Lo novedoso de la metodología es el conjunto de reglas en las que se basa su método de trabajo, a continuación se enumeran las fases de la metodología explicadas de la manera más sencilla para su mejor comprensión.

5.1.1 Fases de la programación extrema

5.1.1.1 Planificación

5.1.1.1.1 Historias de usuarios

Las historias de usuario tienen el mismo propósito que los casos de uso, pero no son lo mismo. Éstas son redactadas por los clientes y en ellas indican como ven y perciben las necesidades del sistema. Deben de ser descripciones cortas y en lenguaje de usuarios finales, es decir, sin utilizar terminología técnica. Tampoco debemos confundir las historias de usuarios con la especificación de requisitos, porque las historias solamente proporcionaran detalles que servirán para la estimación de riesgos y el tiempo necesario para implementar dicha historia.

5.1.1.1.2 Plan de entregas

Las historias de usuario servirán para crear el plan estimado de entrega. El plan de entregas se usará para crear los planes de iteración para cada iteración. El plan de entregas es creado tanto por clientes como desarrolladores. Para formar el plan de

entregas cada historia de usuario es evaluada para desarrollarse en tiempo ideal, el cliente debe de ordenar todas las historias en orden de prioridad o necesidad. De esta manera, el plan de entregas es formado en función de dos parámetros: tiempo de desarrollo ideal y grado de importancia para el cliente.

5.1.1.1.3 Velocidad del proyecto

La velocidad del proyecto es una métrica de cuán rápido se está desarrollando el sistema. Esta métrica se usa para determinar cuántas historias de usuario pueden ser implementadas antes de una fecha dada, o cuánto tiempo es necesario para llevar a cabo un conjunto de historias.

5.1.1.1.4 Crear iteraciones

Una iteración es un periodo de tiempo de desarrollo e idealmente oscila entre una y tres semanas. Al principio de cada iteración se debe de convocar una reunión para trazar el plan de iteración correspondiente. Se usará la velocidad del proyecto para determinar si una iteración está sobrecargada, es decir verificar si la suma de los días que costará desarrollar todas las tareas de la iteración no sobrepasa la velocidad del proyecto en su iteración anterior.

Si se determina que la iteración está sobrecargada, el cliente deberá decidir qué historias de usuario retrasar a una iteración posterior para quitarle carga a la iteración, y por el contrario, si la iteración está holgada se agregarán historias de usuario hasta que la iteración tenga una carga adecuada.

5.1.1.1.5 El plan de iteración

El plan de iteración consiste en seleccionar las historias de usuario que, de acuerdo al plan de entregas, corresponden a esta iteración. También se eligen qué pruebas de aceptación que fallaron anteriormente serán corregidas. Cada historia de usuario se transformará en tareas de desarrollo. Cada tarea de desarrollo corresponderá a un periodo ideal de uno a tres días de ejecución.

5.1.1.1.6 Rotación de personal

Es necesario hacer una rotación constante de los programadores para evitar que los mismos se conviertan en cuellos de botella, porque si sólo un determinado número de programadores trabaja un área específica del sistema, existe un enorme riesgo si son eliminados del proyecto por cualquier circunstancia. La rotación de personal permite que todos los programadores conozcan el sistema completo y es posible realizar un reparto más equitativo del trabajo.

5.1.1.1.7 Reuniones de seguimiento

Las reuniones de seguimiento son necesarias para mantener una comunicación constante entre las diferentes partes que intervienen en el proyecto. Es recomendable que sean frecuentes, de poca duración y, de ser posible, interactuando con el sistema para que el cliente identifique posibles detalles no especificados con anterioridad y puedan ser corregidos a tiempo, para que el impacto del cambio sea mínimo.

5.1.1.2 Diseño

5.1.1.2.1 Simplicidad

Es muy importante que nuestro diseño, tanto de datos como de interfaz, sea lo más sencillo posible, porque siempre costará menos tiempo de implementar un diseño sencillo que uno complejo. Si alguna parte de la implementación resulta especialmente compleja, deberemos rediseñarlo en partes más pequeñas para que cualquier cambio y modificación sean mucho más sencillos de implementar.

5.1.1.2.2 Metáfora del sistema

La idea principal de crear una metáfora para el sistema es para que todo el mundo pueda contar a cerca de cómo funciona. También nos permitirá mantener la coherencia de nombres de todo lo que se va a desarrollar, aunque ésta está más enfocada a la nomenclatura a utilizar (nombres de variables, constantes, etc.) es útil para que cualquier programador pueda entender la relación existente entre los distintos componentes del sistema.

5.1.1.2.3 Tarjetas CRC

Las tarjetas de Clase Responsabilidad y Colaboración (CRC) nos permitirán que el equipo completo participe en el diseño del sistema. Una tarjeta CRC representa un objeto dentro del sistema, el nombre de la clase se coloca a modo de título en la tarjeta, las responsabilidades asociadas se colocan a la izquierda, y las clases que están involucradas en cada responsabilidad se colocan a la derecha, en la misma línea correspondiente.

La siguiente figura muestra un ejemplo de tarjeta CRC.

Figura 33. Tarjeta clase responsabilidad colaboración (CRC)

Nombre de la clase	
Responsabilidad	**Colaboración**
$Responsabilidad_1$	$Clase_1$
$Responsabilidad_2$	$Clase_2$
$Responsabilidad_n$	$Clase_n$

Fuente: Roger S. **Presuman. Ingeniería del *software*** . Página 370.

5.1.1.2.4 No se añadirá funcionalidad en las primeras etapas

Evitaremos añadir funcionalidad extra al sistema. Incluso si conocemos exactamente cómo implementarlas, debemos colocar únicamente la indicada en el plan de iteración. Es decir, debemos centrarnos en la o las tareas que rige el plan de iteración y no perderemos tiempo en programar un código que no se sabe si será utilizado posteriormente.

5.1.1.2.5 Reaprovechar cuando sea posible

Al eliminar un código redundante, eliminamos funcionalidad inútil, y reciclamos código. Esto dentro del ciclo de vida de un proyecto ahorra tiempo e incrementa la calidad. También mantenemos el código limpio y fácil de comprender, modificar y ampliar. Esto puede resultar un poco costoso al principio, pero resulta fundamental a la hora de realizar diseños futuros.

5.1.1.3 Desarrollo

5.1.1.3.1 Disponibilidad del cliente

Uno de los requerimientos principales de la programación extrema es que el cliente debe de estar disponible, no sólo para colaborar con los programadores sino también para ser parte del equipo de desarrollo. La participación del cliente es indispensable porque debe de colaborar en la realización de los *tests* o pruebas que se le realizaran al sistema, porque deberá comprobar los resultados obtenidos de las pruebas y deberá decidir si son satisfactorias de acuerdo a sus necesidades o es necesario hacer modificaciones antes de avanzar a la siguiente iteración.

5.1.1.3.2 Se debe escribir el código de acuerdo a los estándares

Todo el código que se programe deberá estar apegado estrictamente a los estándares de programación indicados por el jefe de proyecto. Entendemos por estándares de programación a la nomenclatura utilizada, por ejemplo, nombres de variables, nombres de procedimientos y funciones, etc. Esto es con el objetivo de que cada programador puede implementar una solución distinta a un problema y ser todas funcionales. Si todas estas soluciones tienen el mismo estándar de programación, cualquier programador debe ser capaz de entender las distintas soluciones.

5.1.1.3.3 Desarrollar la unidad de pruebas primero

Si las distintas pruebas que se le realizarán al código son elaboradas antes de implementarlos, el tiempo de desarrollo será menor porque de antemano se conoce en qué puntos el programa no debe de fallar. Esta característica es el pilar básico sobre el cual se sustenta la metodología de la programación extrema.

5.1.1.3.4 Todo el código debe programarse por parejas

Todo el código deberá de ser desarrollado en parejas que trabajarán de manera coordinada. Esta característica se basa en que la pareja posee conocimientos similares en cuanto a lo que deben desarrollar. Una persona se encarga de pensar la estrategia con que se va a enfrentar una tarea específica y la otra persona se encargará de pensar cómo implementarlo de manera óptima. En otras palabras, uno piensa y el otro programa. Es importante que la pareja intercambie de roles constantemente para no caer en la monotonía.

5.1.1.3.5 Una pareja se encargará de integrar el código

Al integrar el código con otros módulos, pueden aparecer problemas debido a que no han sido evaluados todavía. Las unidades de *test* serán las que se encargarán de verificar la corrección de dichos módulos. Los *tests* deben ser completos, en el sentido de que un fallo en el mismo podría derivar que determinados errores pasaran inadvertidos posteriormente.

5.1.1.3.6 Integrar frecuentemente

Cada pareja de programadores deberá actualizar su código una vez que el éxito en los *test* de prueba corresponde al 100%, es decir que no existe ningún problema. De esta manera, todos los programadores trabajarán con la última versión, y esta actualización constante nos permitirá una detección rápida de fallas en los módulos a consecuencia de incompatibilidad en los mismos.

5.1.1.3.7 Todo el código es común a todos

Si el código es común a todos los desarrolladores, entonces podremos efectuar la rotación de personal sin ningún problema, y cualquier programador podrá modificar el código para agregar alguna funcionalidad o eliminar fallos. Y de esta manera la responsabilidad del funcionamiento del sistema recaerá sobre el equipo completo y no sólo a un grupo de programadores.

5.1.1.3.8 Dejar las optimizaciones para el final

Las optimizaciones del código se llevarán a cabo una vez terminadas las iteraciones, para evitar perder el tiempo en una de éstas intermedia que nos atrase las siguientes iteraciones. De esta manera tendremos un sistema completamente funcional y será mucho más fácil realizar las posibles optimizaciones, si son necesarias, sin perder la funcionalidad del sistema, como sucede a veces si se optimiza por partes y posteriormente se integra al sistema final.

5.1.1.4 Pruebas

5.1.1.4.1 Todo el código debe ir acompañado de su unidad de pruebas

Como se mencionó anteriormente, las pruebas constituyen uno de los pilares de la metodología de la programación extrema, es indispensable que después de efectuar una modificación al código evaluemos nuevamente para verificar que el cambio no induzca a un cambio en la funcionalidad. No debemos olvidar que si agregamos funcionalidad a nuestro código, deberemos rediseñar las pruebas para que la probabilidad de que exista un fallo por incoherencia entre código y prueba sea mínima.

5.1.1.4.2 ¿Qué hacer cuando falla una prueba?

Si se detecta un fallo al realizar las pruebas, deberemos de crear inmediatamente una nueva unidad, para que la localización de la falla sea mucho más fácil para los programadores. El objetivo de crear una nueva unidad de prueba es aislar la falla, depurarla y corregirla directamente.

5.1.1.4.3 Ejecutar pruebas de aceptación

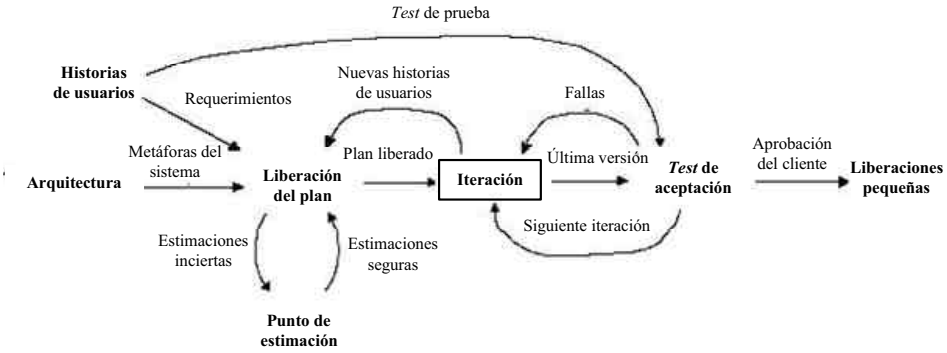
Las pruebas de aceptación son creadas tomando como base las historias de usuarios, porque durante una iteración la historia de usuario seleccionada en la planificación de la iteración correspondiente, se convertirá en una prueba de aceptación. El cliente especifica los aspectos que se deberán de evaluar si la historia de usuario ha sido correctamente implementada. Una prueba de usuario podrá tener tantas pruebas de aceptación como sean necesarias para garantizar su perfecto funcionamiento.

Una historia de usuario no se considera completa hasta que ha superado todas sus pruebas de aceptación, esto implica que deberá crearse una nueva prueba de aceptación por cada iteración que se realice, de lo contrario se considera que no se ha realizado ningún progreso. Los resultados de las pruebas deberán ser conocidas por todos los involucrados en el proyecto, con la mayor brevedad posible, para que los programadores puedan realizar con la mayor rapidez posible los cambios necesarios.

Las pruebas de aceptación son conocidas también como pruebas de funcionalidad, y de manera análoga constituyen las pruebas de la caja negra en la metodología tradicional.

5.1.2 Esquema de la programación extrema

Figura 34. Esquema de la programación extrema (XP)



Fuente: ***Extreme programming: a gentle introduction***. http://www.extremeprogramming.org. (27/03/2003)

5.2 Modelo basado en ensamblaje de componentes

El modelo de desarrollo basado en componentes posee muchas de las características del modelo en espiral, con la diferencia de que configura aplicaciones desde componentes o clases existentes de *software*.

Dado que esta metodología está enfocada en la reutilización de *software* para el desarrollo de las aplicaciones, para realizar la clasificación de las clases candidatas que no son más que posibles componentes que podemos usar en el proyecto, es necesario que entendamos claramente el concepto de ***"Ingeniería del dominio"*** que es base fundamental para esta metodología.

A continuación se explicará este término, de la manera más sencilla posible, para comprender el concepto y su importancia no sólo para esta metodología, sino también para las metodologías orientadas a objetos.

5.2.1 Ingeniería del dominio

Se refiere específicamente al análisis orientado a objetos que se realiza en un nivel medio de abstracción, es decir, sin muchos ni pocos detalles, únicamente los necesarios.

5.2.1.1 Análisis del dominio

Es identificar, analizar y afinar requisitos comunes de un dominio de aplicación específico para que estos puedan ser reutilizados en múltiples proyectos que pertenezcan al mismo dominio de aplicación. El objetivo del análisis del dominio es encontrar o crear clases o componentes que serán utilizados por aplicaciones similares.

5.2.1.2 Definición del dominio

Se refiere al aislamiento del área de interés, tipo de sistema o categoría de nuestro proyecto, y consiste en extraer los elementos orientados a objetos como los no orientados. Los primeros deben de incluir especificaciones, diseños y código para clases de aplicaciones ya existentes; por ejemplo, interfases graficas, menús, acceso a base de datos, etc.

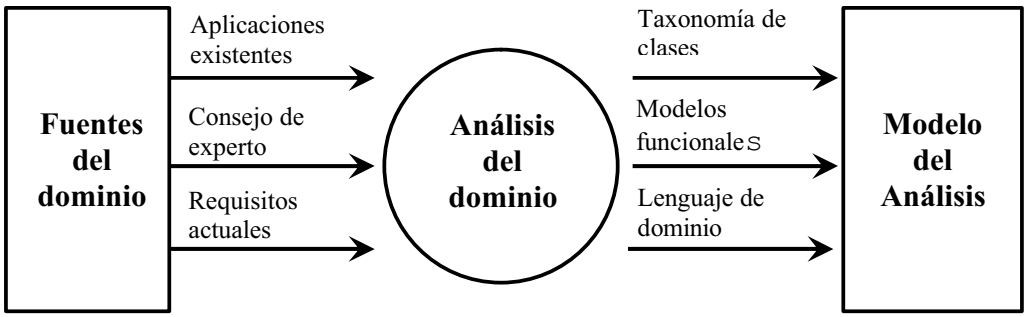
5.2.1.3 Clasificación de los elementos

Se refiere a organizar los elementos definidos en categorías y establecer las características generales de la categoría.

Es recomendable sugerir un esquema de clasificación y definir alguna convención para la nomenclatura de los elementos y, de ser posible, establecer jerarquías de clasificación si se considera necesario.

La siguiente figura muestra un esquema general para realizar de manera correcta el análisis del dominio para una aplicación específica.

Figura 35. Entradas y salidas para el análisis del dominio



Fuente: Roger S. Pressman. **Ingeniería del Software**. Página 365.

5.2.2 Fases del ensamblaje de componentes

5.2.2.1 Análisis y diseño arquitectónico

El análisis a efectuar debe enfocarse en la metodología orientada a objetos y en el análisis del dominio para delimitar nuestra aplicación a un área específica. Por medio del diseño arquitectónico se sintetizarán representaciones de las estructuras de datos que se utilizarán, estructura del programa, características de la interfaz y los detalles de los distintos procesos, basándonos en los requisitos establecidos.

5.2.2.2 Cualificación de componentes

Nos asegura que un componente candidato realizará una función necesaria, además de adaptarse al estilo de arquitectura especificado para el sistema y poseer las características de calidad necesarias para la aplicación. Los siguientes son algunos factores importantes que se deben tomar en cuenta al efectuar la cualificación de componentes:

a) La interfaz de programación para las aplicaciones (API).
b) Las herramientas de desarrollo e integración necesarias para cada uno de los componentes.
c) Requisitos de ejecución y recursos utilizados por cada uno de los componentes.
d) Funciones de seguridad, por ejemplo encriptamiento de datos, protocolos de autenticación, control de accesos, etc.
e) Manipulación de excepciones o errores en tiempo de ejecución, etc.

5.2.2.2.1 Adaptación de componentes

Un componente tiene un alto grado de adaptación si se cumplen las siguientes condiciones:

a) Existen métodos de gestión de recursos para todos los componentes de la biblioteca.
b) Existen actividades comunes para todos los componentes.
c) Existen interfaces dentro de la arquitectura con un entorno externo consecuente para el sistema.

Sin embargo, después de haber cualificado los componentes para su uso dentro de una arquitectura de aplicación, existe la posibilidad de que surjan conflictos en una o más de las áreas anteriores. Para minimizar estos conflictos, se debe utilizar alguna técnica de adaptación, llamadas también de encubrimiento de componentes. Las técnicas más comunes son las siguientes:

a) **Encubrimiento de caja blanca:** se aplica cuando tenemos a nuestra disposición el código fuente del componente para realizar las modificaciones pertinentes.
b) **Encubrimiento de caja gris:** es aplicable cuando la biblioteca de componentes proporciona un lenguaje de extensión que permita eliminar o enmascarar los conflictos existentes.
c) **Encubrimiento de caja negra:** necesita de un preprocesamiento o postprocesamiento en la interfaz de componentes para eliminar o enmascarar conflictos. Si no es posible enmascarar adecuadamente el componente, se debe diseñar un componente personalizado.

5.2.2.2.2 Composición de componentes

En esta etapa se ensamblan los componentes cualificados, adaptados y diseñados para la arquitectura establecida para una aplicación. Para hacer esto es necesario establecer una infraestructura en donde los componentes estarán unidos a un sistema operacional, que proporcionará un modelo para la coordinación de componentes y servicios específicos que realizarán tareas comunes.

Para realizar una composición o ensamblaje de componentes es necesario tomar en cuenta los siguientes factores:

a) **Modelo de intercambio de datos:** son mecanismos que capacitan a los usuarios y a las aplicaciones para interactuar y transferir datos y que deben estar definidos en todos los componentes reutilizables.
b) **Automatización:** facilita la interacción entre componentes reutilizables se deben de definir macros, guiones o utilizar las distintas herramientas que faciliten esta tarea.
c) **Almacenamiento estructurado:** almacenar la información de manera estructurada, por ejemplo, separar datos homogéneos y datos no homogéneos. Por datos homogéneos se entiende información, y por datos no homogéneos los que no son de tipo primitivo como datos gráficos, de voz, vídeo, etc., que requieren un procesamiento especial.
d) **Modelo de objetos subyacente:** asegura que los componentes desarrollados en distintos lenguajes de programación y distintas plataformas puedan ser ínter operables.

5.2.2.3 Ingeniería de componentes

La ingeniería de componentes fomenta la utilización de componentes de *software* ya existentes. Para aprovechar al máximo las ventajas de esta metodología, debemos tener en cuenta lo siguiente:

a) **Datos estándar:** se debe investigar el dominio de la aplicación e identificar de manera precisa las estructuras de datos estándar. Por ejemplo, archivos, bases de datos, etc.
b) **Protocolos de interfaz estándar:** se refiere a la naturaleza de las interfaces intra modulares, diseño de interfaces técnicas externas, e interfaz hombre máquina.
c) **Plantillas de programa:** el modelo de una estructura se utiliza como plantilla para el diseño arquitectónico de otro programa.

5.2.2.4 Actualización de componentes

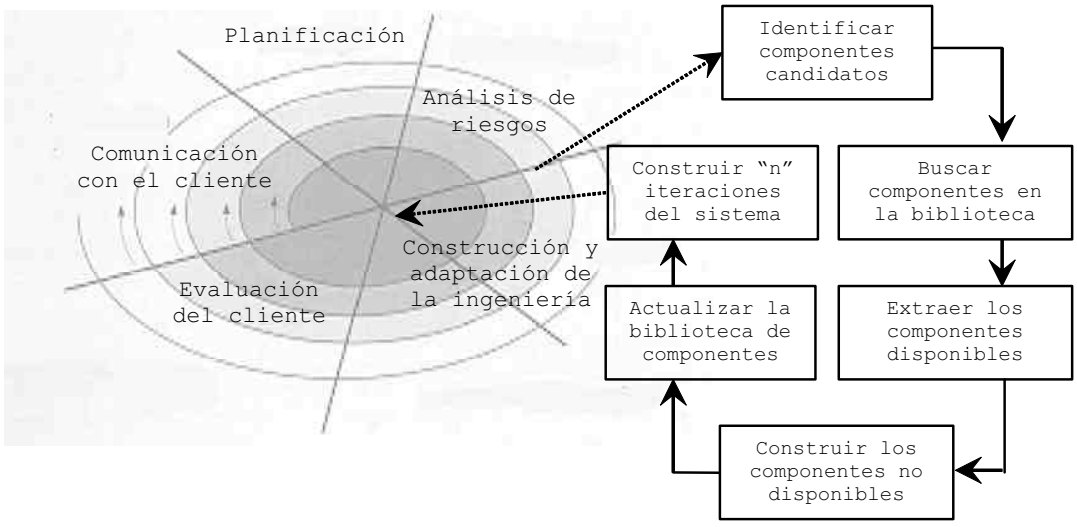
Se deben actualizar los componentes del sistema de manera constante, de la misma manera que se realiza en la programación extrema. Para más detalles véase la sección 5.1.1.3.6 de este mismo capítulo.

5.2.2.5 Comprobación

La comprobación no es más que la verificación del funcionamiento del sistema de acuerdo a los requisitos establecidos. Para realizarla se utilizan las estrategias de prueba de *software* convencionales como lo son las pruebas de unidad, de integración, de validación, del sistema, etc.

5.2.3 Esquema del modelo ensamblaje de componentes

Figura 36. Esquema del modelo ensamblaje de componentes



Fuente: Roger S. Pressman. **Ingeniería del *software*** . Página 28.

5.3 Modelo en V

El modelo en V es una especie de combinación entre el modelo tradicional, es decir, el modelo en cascada y el ensamblaje de componentes. Sin embargo, este modelo da importancia a la simetría y jerarquía que adoptan las distintas partes del sistema durante su tiempo de desarrollo.

Las principales consideraciones se basan en la introducción de actividades de planeación y ejecución de pruebas en cada etapa como parte del desarrollo. En este modelo se pueden establecer un número finito de fases que se considere necesario, y su evolución o avance es en forma de V.

En donde el eje horizontal representa avance en el desarrollo y el eje vertical se corresponde al nivel de detalle con el que se trabaja en cada fase. Las fases iniciales en la rama izquierda son descendientes y se van descomponiendo en elementos cada vez más sencillos hasta llegar a las sentencias del lenguaje de programación o componentes reutilizables, y la rama derecha es ascendente e indica las pruebas o *test* que se deben efectuar en cada fase, antes de avanzar a la siguiente.

5.3.1 Fases del modelo en V

5.3.1.1 Rama izquierda

5.3.1.1.1 Captura de requisitos

Es la etapa inicial del modelo, y es la encargada de identificar todos los requisitos del sistema, para llevar a acabo esta tarea se pueden utilizar diversas técnicas incluidas en el capítulo 2, incisos 2.1.1.1 entrevistas a usuarios, 2.1.1.2 cuestionarios de preguntas abiertas y cerradas y 2.1.1.3 observación directa del sistema.

Esto no significa que debamos cerrarnos y utilizar únicamente los métodos convencionales, porque podríamos hacer esta captura de requisitos por métodos como la redacción de historias de usuarios incluida en este mismo capítulo en el inciso 5.1.1.1.1, comparar los resultados y obtener resultados más precisos.

5.3.1.1.2 Análisis de requisitos

Una vez obtenidos todos los requisitos del sistema, en la fase de análisis debemos construir la estructura del nuestro. Identificar los distintos módulos que formarán nuestro sistema y las distintas relaciones existentes entre los datos y diversos módulos.

5.3.1.1.3 Diseño de la arquitectura

Como se indicó en la sección 5.2.2.1, el diseño arquitectónico nos proporciona una visión global del sistema que se desea construir, por lo tanto debemos describir la estructura y organización de los componentes, propiedades y conexión entre ellos; agrupa un conjunto de actividades que nos conducirán a un modelo completo del diseño de *software*.

5.3.1.1.4 Diseño de componentes

En esta fase se diseñan los componentes que no existen en nuestra librería de componentes, o bien diseñar las modificaciones o adaptaciones que debemos efectuar a los componentes existentes. Para mayor información véase secciones 5.2.2.2.1 y 5.2.2.2.2.

5.3.1.1.5 Codificación de componentes

Se refiere a la construcción del componente de acuerdo a su diseño respectivo. Al construir el componente debemos tomar en cuenta factores como el lenguaje de programación a utilizar, interfaz de programación, depuración de errores, etc. Porque mientras más amigable sea la herramienta de trabajo, serán mucho más fácil sus actualizaciones futuras.

5.3.1.2 Rama derecha

5.3.1.2.1 Prueba unitaria

La prueba de unidad se enfoca específicamente en la verificación de la menor unidad de diseño del *software*, por ejemplo un componente o módulo.

Una prueba de unidad básicamente consiste en probar la interfaz del módulo o componente para asegurar que los datos permanecen íntegros y no se pierden o alteran en los distintos procesos involucrados. Se prueban las condiciones límite para asegurar que los límites de procesamiento han sido establecidos correctamente; se prueban los caminos independientes o básicos, y también todos los caminos de manejo de errores. Para más información véase la sección de verificación o pruebas incluidas en el capítulo 2, incisos 2.1.5.2 pruebas de la caja blanca y 2.1.5.3, pruebas de la caja negra, respectivamente.

5.3.1.2.2 Integración de sistemas y subsistemas

Consiste en verificar el perfecto funcionamiento primero de los subsistemas, posteriormente la integración de los distintos subsistemas que formaran nuestro sistema. Para realizar esta tarea utilizamos diversas técnicas de integración, por ejemplo, la integración descendente y ascendente que se describen a continuación.

a) **Integración descendente:** consiste en integrar todos los módulos del sistema avanzando hacia abajo, de acuerdo a la jerarquía de control establecida; se inicia por el módulo principal y los módulos subordinados se van integrando uno a uno al módulo principal. De a cuerdo a nuestra jerarquía de control, esta integración la podemos hacer de dos maneras distintas, primero integrando en profundidad o bien primero en ancho; el enfoque que utilicemos no debe alterar la funcionalidad del sistema.
b) **Integración ascendente:** el enfoque de integración de esta técnica es opuesta a la anteriormente descrita, porque inicia su integración desde los módulos o componentes atómicos, es decir, los que están en el nivel más bajo de nuestra jerarquía de control. La idea es integrar un módulo de nivel inferior, integrarlo al nivel superior, continuar este proceso hasta integrarlo al módulo principal.

5.3.1.2.3 Pruebas de aceptación

Una vez integrado el sistema en su totalidad, debemos realizar una serie de pruebas para verificar que se han integrado adecuadamente todos los elementos del sistema. Es importante hacer notar que cada una de las siguientes pruebas tiene un propósito diferente y es necesario realizarlas a todos los sistemas que construyamos.

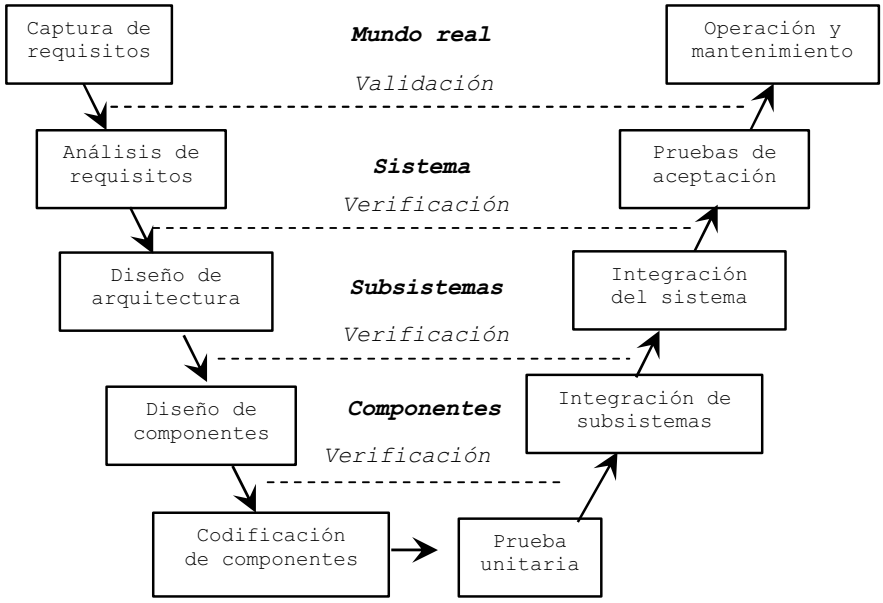
a) **Prueba de recuperación:** en algunas ocasiones un sistema debe ser tolerante a fallos, es decir que factores externos no deben de hacer que se detenga el funcionamiento del sistema. En otros casos se debe recuperar de un fallo en un determinado periodo de tiempo para que no se genere un daño, generalmente económico.
b) **Prueba de seguridad:** cualquier sistema de información debe garantizar la seguridad de la información y la confidencialidad de los mismos. Esta prueba verifica que los mecanismos de protección incorporados en el sistema lo protegerán de accesos impropios. Es importante hacer notar que ningún sistema es seguro en un 100%, sin embargo, debemos asegurar que el costo de violar la seguridad del sistema sea mayor que el valor de la información obtenida.
c) **Prueba de resistencia:** está diseñada para probar el sistema en situaciones anormales. Este tipo de prueba se ejecuta de manera que el sistema demande recursos en cantidad, frecuencia o volúmenes anormales. El objetivo de esta prueba es encontrar los valores en donde el sistema se vuelve inestable o degrada en su rendimiento.
d) **Prueba de rendimiento:** estas pruebas son necesarias para los sistemas de tiempo real y empotrados incluido en el capítulo 1, sección 1.3 (aplicaciones del *software)*, incisos 1.3.2 *software* de tiempo real y 1.3.6 *software* empotrado, respectivamente; y está diseñada para probar el rendimiento del *software* en tiempo de ejecución dentro del contexto de un sistema integrado. Este tipo de prueba se lleva a cabo durante todos los pasos, se debe asegurar el rendimiento de los módulos individuales, pero hasta que no están verificados todos los elementos del sistema no se puede asegurar realmente su rendimiento.

5.3.1.2.4 Operación y mantenimiento

Una vez instalado el sistema, se le debe dar mantenimiento de acuerdo a las necesidades de cambio como se explica en el capítulo 2, sección 2.1.6 (mantenimiento y documentación externa), inciso 2.1.6.1 (mantenimiento).

5.3.2 Esquema del modelo en V

Figura 37. Esquema del modelo en V



Fuente: **Ciclo vida *software***. http://mural.uv.es/givaro/modulo/Ciclo.htm. (20/08/2002)

6. ANÁLISIS COMPARATIVO

En el presente capítulo se realizará un análisis comparativo entre las distintas metodologías de desarrollo de *software*, basándonos en los distintos parámetros que se deben tomar en cuenta en la elaboración de un proyecto de *software*. El objetivo de este análisis es obtener una tabla en donde de acuerdo a los valores mostrados, para las distintas metodologías y factores se pueda determinar la metodología óptima que se adapte más a las necesidades del proyecto.

6.1 Factores a considerar

6.1.1 Disponibilidad de recursos o factibilidad del proyecto

La viabilidad y el análisis de riesgo están relacionadas de muchas maneras, por ejemplo, si el riesgo del proyecto es alto, la viabilidad de producir *software* de calidad se reduce considerablemente. Por lo tanto, cada recurso que se utilice se debe especificar mediante las siguientes cuatro características:

a) **Descripción del recurso:** una breve descripción del recurso que se va a utilizar, indicando claramente si se trata de un recurso humano, de *software* o hardware.
b) **Informe de disponibilidad:** nos permitirá determinar si el recurso a utilizar está a su plena disposición a lo largo del proyecto.
c) **Fecha cronológica de inicio:** fecha de inicio tanto estimada como real del inicio de actividades del proyecto.
d) **Fecha cronológica de fin:** fecha de culminación del proyecto, tanto real como estimado.

Obsérvese que para las fechas cronológicas se utilizan dos, la fecha real y la fecha estimada; se hace de esta manera para desde un inicio llevar control del avance del proyecto y sus posibles atrasos. Así se determinarán de manera óptima las actividades que causan atraso e las fechas de entrega y se reasignarán recursos en donde sea necesario para cumplir a cabalidad con las fechas planificadas.

Durante el desarrollo de un proyecto de *software* debemos concentrar nuestra atención en cuatro áreas principales, porque todo el desarrollo del proyecto gira entorno a ellas. Es necesario recalcar que ninguna de éstas es excluyente una de otras, porque todas son necesarias e indispensables en alguna medida para la puesta en marcha del proyecto. Estas áreas son las que a continuación se enumeran:

6.1.1.1 Viabilidad técnica

Por factibilidad técnica se comprende todos nuestros recursos tantos de *software* como de hardware, con los que se cuenta o se requiere. Se debe evaluar si los recursos técnicos pueden mejorarse para poder cumplir con necesidades futuras, debido a la rapidez con la que la tecnología avanza constantemente, y si es necesario añadir nuevos recursos, aunque esto repercute directamente en la factibilidad económica.

6.1.1.2 Viabilidad económica

Determinar la factibilidad económica con la que se cuenta en los aspectos básicos tales como: costo del tiempo del equipo de desarrollo, costo de hardware, costo del *software* y costo del tiempo propio.

El estudio de factibilidad económica involucra un análisis beneficio – costo por medio del cual se determina si el beneficio obtenido con el proyecto es mayor al costo del mismo; de no ser así se deben buscar otras alternativas más factibles económicamente porque si el costo es mayor que el beneficio, el proyecto es básicamente no factible llevarlo a cabo.

6.1.1.3 Viabilidad operacional

Nos referimos a la disponibilidad del recurso humano que está disponible para la realización del proyecto. Es importante determinar el número exacto de personas que participarán a lo largo del proyecto, porque está relacionado directamente con el tiempo de desarrollo del sistema.

Es recomendable mantener en la medida de lo posible a los mismos participantes en el proyecto, porque el cambio o sustitución de algunos de sus elementos, aunque el número de participantes es el mismo, la continuidad del elemento sustituido no puede ser la misma y esto provocará un retraso en la fecha de culminación del proyecto.

6.1.1.4 Viabilidad legal

Consiste en determinar cualquier infracción, violación o responsabilidad legal en que se podría incurrir por el desarrollo del sistema. Por ejemplo, el uso de *software* sin el licenciamiento necesario o en otras palabras la utilización de *software* pirata para la elaboración del sistema.

Esta área se puede omitir o descartar por completo únicamente si el *software* a desarrollar será por medio de herramientas de desarrollo de licencia *software* libre entiéndase licencias GPL (*General Public Lincence*) o *Copyleft*.

La utilización de *software* libre involucra tanto ventajas como desventajas en el desarrollo de un proyecto. Entre las principales ventajas se encuentra el costo, porque las herramientas de desarrollo son gratuitas y se pueden obtener fácilmente, sin embargo, la principal desventaja es el soporte técnico con que cuentan algunas de estas herramientas.

Otras ventajas son las siguientes:

a) Libertad para ejecutar el programa, con cualquier propósito.
b) Libertad para modificar el programa para adaptarlo a nuestras propias necesidades, porque para que esta libertad sea efectiva en la práctica, se debe tener acceso al código fuente, cosa que no sucede con algunos programas, por ejemplo, los diversos *software* ofrecidos por la empresa *Microsoft*.
c) Libertad para redistribuir copias, tanto gratis como por medio de un cánon.
d) La libertad para distribuir versiones modificadas del programa, de tal manera que la comunidad pueda beneficiarse con sus mejoras.

La siguiente tabla muestra algunas de las herramientas de *software* libre que pueden ser utilizadas para el desarrollo de un proyecto:

Tabla VII. Software libre

AREA DE APLICACIÓN	*SOFTWARE*
Paquetes de oficina	➢ OpenOffice ➢ StarOfffice
Compresores de archivo	➢ Gnozip ➢ ARK ➢ Linzip
Bases de datos	➢ MySQL ➢ PostgreSQL ➢ AdaBase
Lenguajes de programación	➢ PHP ➢ Perl ➢ ASP ➢ Java
Sistema operativo	➢ Linux

6.1.2 Complejidad del proyecto

Al planificar un proyecto se deben obtener estimaciones del esfuerzo humano requerido, de la duración cronológica del proyecto y del costo. En la mayoría de los casos, estas estimaciones se hacen en base a experiencias pasadas como única guía, sin embargo, en algunos casos puede que la experiencia no sea suficiente.

Para tener un cálculo más exacto, se debe utilizar técnicas o métricas de estimación que es una forma de resolución de problemas en donde descomponemos el problema en un conjunto de pequeños problemas, y asignar a cada uno de ellos un tiempo específico.

Existen distintas métricas que podemos utilizar para estimar la complejidad del proyecto, por ejemplo, líneas de código ó LDC, puntos de función o PF o el modelo constructivo de costo, comúnmente llamado COCOMO. La técnica que se va a explicar a continuación es la de puntos de función porque es un método para cuantificar el tamaño y la complejidad de un sistema *software* en términos de las funciones de usuario que éste desarrolla o desarrollará.

Además proporciona ventajas sobre las otras técnicas, por ejemplo:

a) Son independientes del lenguaje, herramientas o metodologías utilizadas en la implementación; es decir, no tienen que considerarse lenguajes de programación, sistemas de administración de bases de datos, hardware, o cualquier otra tecnología de procesamiento de datos.
b) Pueden ser estimados a partir de la especificación de requisitos o especificaciones de diseño, haciendo posible de este modo la estimación del esfuerzo de desarrollo en etapas tempranas del mismo. Además, como los puntos de función están íntimamente relacionados con la declaración de requisitos, cualquier modificación a ésta puede ser reflejada sin mayor dificultad en una reestimación.
c) Están basados en una visión externa del usuario del sistema. Los usuarios no técnicos del *software* poseen un mejor entendimiento de lo que los puntos de función están midiendo.

6.1.2.1 Puntos de función

Está orientada a la función que utiliza la funcionalidad entregada por la aplicación como un valor de normalización. Los puntos de función se derivan de una relación de las medidas del dominio de información del *software* y las estimaciones de complejidad del mismo.

Se determinan cinco características o dominios de información y se les asigna una medida apropiada. Los dominios a determinar son los siguientes:

a) **Número de entradas de usuario:** se cuenta cada entrada de información proporcionada por el usuario en las distintas interfaces o pantallas de ingreso de datos.
b) **Número de salidas de usuario:** se cuenta cada una de las salidas que el sistema proporciona al usuario, por ejemplo, informes, pantallas, mensajes de error, etc.
c) **Número de peticiones de usuario:** es una entrada interactiva que da como resultado alguna salida al usuario, por ejemplo, consultas.
d) **Número de archivos:** se cuenta cada archivo maestro lógico o grupo lógico de datos que puede ser una o varias tablas de una base de datos o bien un archivo independiente.
e) **Número de interfaces externas:** se cuentan todas las interfaces que se utilizan para transmitir información a otro sistema.

A cada uno de los dominios anteriores se les asigna un valor de complejidad que será un valor entero no negativo relacionado a un grado de complejidad, por ejemplo simple, mediano y complejo. Aunque el grado de complejidad se deja a criterio del jefe de proyecto ya que se pueden determinar niveles intermedios o superiores dependiendo del proyecto.

Para estimar el valor de puntos de función, se utiliza la siguiente fórmula:

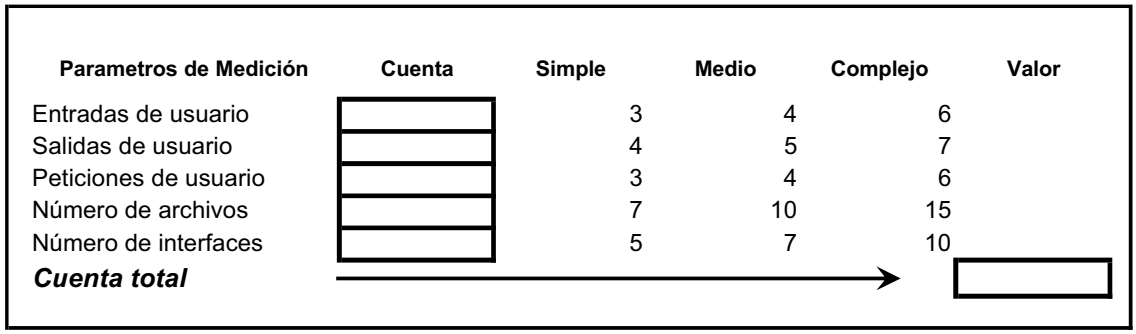
$$\mathbf{PF = Cuenta\ Total\ x\ [0.65+0.01\ x\ 6\ x\ \Sigma(F_i)]}$$

En donde cuenta total es la suma de todas las entradas obtenidas de la tabla de la figura 38 y F_i(i = 1 ... 14) son valores de ajuste de complejidad que se le asigna según el valor obtenido de contestar las siguientes preguntas, se utiliza una escala con rangos desde 0 (no importante o no aplicable) hasta 5 (absolutamente esencial).

1. ¿Requiere el sistema copias de seguridad y de recuperación fiables?
2. ¿Se requiere comunicación de datos?
3. ¿Existen funciones de procesamiento distribuido?
4. ¿Es crítico el rendimiento?
5. ¿Se ejecutará 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 u operaciones?
8. ¿Se actualizan los archivos maestros de forma interactiva?
9. ¿Son complejas las entradas, salidas, archivos o 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 las aplicaciones para facilitar los cambios y para ser fácilmente utilizada por el usuario?

La siguiente figura muestra una tabla para estimar el valor de puntos de función para un proyecto de *software*.

Figura 38. Cálculo de puntos de función



Fuente: Roger S. Presuman. **Ingeniería del *software***. Página 60.

6.1.3 Manejo de la perspectiva de riesgo

El análisis y la gestión de riesgo son un conjunto de pasos que ayudan al equipo de desarrollo de *software* a manejar de manera adecuada la incertidumbre. Ésta no es más que un riesgo potencial que puede o no ocurrir. Independientemente de los daños que pueda causar, es importante identificarlos, evaluar su probabilidad de aparición y evaluar su impacto al proyecto, para establecer un plan de contingencia y enfrentarlo de la mejor manera en caso suceda.

6.1.3.1 Estrategias de riesgo reactivas

Este tipo de estrategias supervisa el proyecto en previsión de posibles riesgos, los recursos se ponen aparte en caso de que pudieran convertirse en problemas reales. En otras palabras, consiste en enfrentar el riesgo directamente sólo cuando éste aparezca.

6.1.3.2 Estrategias de riesgo proactivas

La estrategia proactiva inicia antes de que comience el desarrollo del proyecto; porque antes que nada se identifican los riesgos potenciales, se evalúa su probabilidad de surgimiento y su impacto para establecer una prioridad, según su importancia, con la finalidad de establecer un plan de contingencia para controlar de manera eficaz el riesgo.

Cuando se realiza un análisis de riesgos, es indispensable cuantificar el nivel de incertidumbre y el grado de pérdidas asociado a cada riesgo, además de clasificarlos en las siguientes categorías:

6.1.3.2.1 Riesgos del proyecto

Son los que amenazan al plan, es decir que si los riesgos del proyecto se hacen realidad, es probable que la planificación se retrase y los costos se incrementen.

Por ejemplo, lo siguientes son riesgos del proyecto:

- Cambio, abandono y baja del personal.
- Abandono del cliente, por motivo de nuestra falta de compromiso con respecto al contrato firmado.
- *Hardware* no disponible.

- *Software* no disponible.
- Daño o falla del *hardware*.
- Retraso en el informe de los requisitos y / o diseño del sistema.
- Personal poco calificado.
- Mala elección del lenguaje de programación para la realización del código debido a la falta de conocimiento de los programadores sobre dicho lenguaje.
- Subestimación del trabajo.
- Falta de motivación o moral baja del personal.
- Retraso en la fase de calidad y prueba.
- Retraso en la entrega del proyecto.

6.1.3.2.2 Riesgos técnicos

Son los que amenazan la calidad y la planificación temporal del *software* que hay que desarrollar, porque si un riesgo técnico se vuelve realidad, el desarrollo puede llegar a ser difícil o imposible.

Por ejemplo, los siguientes son riesgos técnicos:

- Cambio de requisitos.
- Mal planteamiento del diseño del sistema.
- Mala optimización de los recursos del sistema, debido al compilador, o a la ineficiencia en la gestión de los recursos del sistema por parte del programador y / o diseñador.
- Mala calidad de los recursos del gestor de base de datos a utilizar.
- Medios no disponibles para entrenar al personal poco cualificado.
- Limitaciones impuestas por el lenguaje de programación a utilizar, por ejemplo, funciones no compatibles o no soportadas.

6.1.3.2.3 Riesgos del negocio

Son los que amenazan la viabilidad del *software* a desarrollar. Estos riesgos a menudo ponen en peligro el proyecto o el producto. Entendemos por producto un sistema que será lanzado al mercado para su comercialización, mientras que por proyecto lo aplicamos más a un proyecto de *software* a la medida. Algunos riesgos de este tipo son los siguientes:

- Desarrollar un producto o sistema que nadie lo va a utilizar, este riesgo se conoce como riesgo de mercado.
- Desarrollar un producto que no se adapta a la estrategia comercial de la empresa, se conoce como riesgo estratégico.
- Desarrollar un producto que el departamento de ventas no sabe como comerciar.
- Perder el apoyo de una gestión debido a cambios de enfoque o a cambios de personal, conocido como riesgo de dirección.
- Perder presupuesto o personal asignado, llamado también riesgo de presupuesto.

6.1.4 Análisis de requerimientos

El análisis de requerimientos es una actividad indispensable y fundamental en todo proyecto de *software*, porque por medio del análisis obtenemos las respuestas a las siguientes interrogantes:

a) ¿Qué se debe de hacer?
b) ¿Cómo se debe de hacer?

Si no se tienen respuestas concretas a estas interrogantes, de ninguna manera puede iniciarse la construcción del sistema.

6.1.5 Volatilidad de los requerimientos

La volatilidad de los requerimientos se refiere al cambio o modificaciones de los requerimientos iniciales a lo largo del desarrollo del proyecto. Es muy improbable que en un proyecto no existan cambios a los requerimientos iniciales, y se debe evitar, en la medida de lo posible, cambios a los requisitos previamente establecidos, no porque afecte al proyecto sino porque puede ser indicio de que el análisis de requerimientos no fue efectuado de manera correcta. Lo anterior da la pauta al surgimiento de nuevos requisitos no establecidos con anterioridad, en cuyo caso afectan al proyecto porque es necesario replantear el análisis y diseño efectuado hasta el momento para adaptarlo a las nuevas necesidades.

6.1.6 Dominio del problema

El dominio del problema está relacionado directamente con la experiencia del analista del sistema. Es recomendable que el análisis y diseño del sistema sea realizado por personas altamente calificadas para esta tarea. Sin embargo, la experiencia no garantiza por completo errores en el análisis y diseño del sistema, únicamente lo minimiza. No hay que olvidar que todos los sistemas son completamente diferentes y no se debe comparar comportamientos con sistemas similares para construir uno nuevo.

6.2 Matriz comparativa de metodologías

La siguiente matriz muestra una comparación entre las distintas metodologías de desarrollo, tomando en cuenta los factores anteriormente descritos que se deben considerar en la implementación de un sistema.

El grado de complejidad del proyecto alto, medio, bajo se debe calcular en base a los requerimientos del proyecto, por algún método de estimación; en este trabajo se

explica el método de puntos de función en este mismo capítulo en la sección 6.1.2.1; y de acuerdo al valor obtenido el jefe de proyecto debe determinar su grado de complejidad. No existe una tabla de valores en donde se indiquen rangos específicos, debido a que cada proyecto es único, y no se debe de comparar con proyectos similares.

6.2.1 Escala de valores

A continuación se define la escala de valores para cada uno de los criterios utilizados en la matriz comparativa:

a) Disponibilidad de recursos

- **Todos:** involucra todos los recursos anteriormente descritos, técnicos, económicos, operacional y legal al inicio del proyecto.
- **Algunos:** involucra todos los recursos, pero no al 100%, desde el inicio del proyecto, porque debido a que son utilizados por metodologías evolutivas. La disponibilidad completa puede obtenerse a lo largo del proyecto; sin embargo, es necesario la plena garantía de obtenerlos en el momento necesario.

b) ***Complejidad del proyecto***

- **Alta:** alto numero de transacciones, operaciones en línea, alto grado de seguridad y confiabilidad, alta disponibilidad (24x7), interacción con sistemas externos, información distribuida; por ejemplo, sistemas de comercio electrónico, cajeros electrónicos, etc.
- **Media:** alto número de transacciones, información y operación centralizadas al final del día, alta disponibilidad; por ejemplo, sistemas bancarios.

- **Baja:** funcionalidad del sistema enfocada en automatización de procesos manuales, cálculos matemáticos no complejos; por ejemplo, sistemas de inventario, contabilidad, etc.

c) ***Análisis de requerimientos***

- **Específico:** funcionalidad del sistema especificada con un alto grado de detalle, operaciones y procesos claramente definidos desde el inicio del proyecto.
- **Medio:** funcionalidad del sistema definida a lo largo del proyecto por medio de iteraciones sucesivas, capturando los requisitos por medio de técnicas como casos de uso o historias de usuarios.
- **Bajo:** funcionalidad del sistema definida en gran parte por componentes reutilizables o afinación de prototipos a lo largo de iteraciones sucesivas.

d) ***Volatilidad de requerimientos***

- **Sí:** significa que la metodología tiene contemplada dentro de sus fases el cambio de requerimientos a lo largo de su desarrollo.
- **No:** lo opuesto a lo anterior; por ejemplo, el modelo en cascada.

e) ***Manejo de la perspectiva de riesgo***

- **Sí:** la metodología tiene contemplada dentro de sus fases los diferentes riesgos que existen en un proyecto de *software* con el objetivo de minimizar el impacto al proyecto ante la presencia del mismo.
- **No:** no tiene contemplado en sus fases de desarrollo planes de contingencia para enfrentar posibles riesgos durante el desarrollo del proyecto.

f) ***Dominio del problema***

- **Alto:** un alto grado de definición y delimitación del sistema, debido a que su desarrollo será llevado a cabo por medio de una serie de etapas sucesivas.
- **Medio:** permite mayor flexibilidad en la definición y delimitación del sistema, debido a que su desarrollo será llevado a cabo por medio de una serie de iteraciones, en donde cada una contiene una serie de etapas por medio de las cuales se va afinando y delimitando el sistema hasta el final de su desarrollo.

Tabla VIII. Matriz comparativa

CRITERIO	CASCADA	ESTRUCTURADO MODERNO	PROTOTIPOS	ESPIRAL	ITERATIVO INCREMENTAL	PROGRAMACIÓN EXTREMA	ENSAMBLAJE DE COMPONENTES	MODELO EN V
Disponibilidad de recursos	Todos	Todos	Algunos	Algunos	Algunos	Algunos	Algunos	Todos
Complejidad del proyecto	Bajo	Bajo	Media	Alta	Alta	Alta	Media	Bajo
Análisis de requerimientos	Específico	Específico	Bajo	Media	Media	Media	Bajo	Específico
Volatilidad de requerimientos	No	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Manejos de la perspectiva de riesgo	No	No	No	Sí	Sí	Sí	Si	No
Conocimiento del dominio del problema	Alto	Alto	Medio	Medio	Medio	Medio	Medio	Alto

6.3 Interpretación de resultados

Como se puede observar en la tabla anterior, aquellas metodologías que requieren todos los recursos disponibles desde un principio del proyecto, son aquellas que podemos considerar como tradicionales. Las podemos aplicar en aquellos proyectos en donde la complejidad y/o tamaño es pequeño.

El grado de análisis de requerimientos es específico y detallado, debido a su poca complejidad, lo que implica un alto grado de conocimiento o dominio del problema; por consiguiente, la volatilidad de requerimientos es poca. Ejemplos de estas metodologías son: modelo en cascada, análisis estructurado moderno y modelo en V.

Por el contrario, si la complejidad del proyecto se puede catalogar como media o alta, las metodologías anteriores no las podemos aplicar de ninguna manera. En este caso debemos de elegir algún modelo evolutivo el cual no nos exige contar con todos los recursos al inicio del proyecto, pero sí nos exige completarlos durante la evolución del mismo cuando estos sean requeridos.

No nos exigen un alto grado o dominio del problema, porque el problema será resuelto por medio de una serie de iteraciones, en donde cada iteración dará como resultado un prototipo el cual será evaluado por el usuario final, y los cambios que surjan serán de afinamiento o mejoras hasta alcanzar la solución final.

Es por esta razón que las metodologías evolutivas poseen un alto grado en la volatilidad o cambios en los requerimientos, pero es importante recalcar que dichos cambios son de afinamiento o mejoras y no por errores debidos a un mal análisis y/o diseño.

Otra característica de las metodologías evolutivas es la incorporación del manejo de la gestión de riesgos, la cual nos ayuda a enfrentar de la mejor manera posible cualquier inconveniente que pueda amenazar la correcta evolución del proyecto en el tiempo. Pero esto no significa que en los proyectos pequeños no existan riesgos, por el contrario, los riesgos están presentes en todo tipo de proyectos. Es importante hacer esta gestión de riesgos en todo tipo de proyectos, sin importar su tamaño y/o complejidad.

CONCLUSIONES

1. Todas las metodologías de desarrollo de *software* tienen como objetivo principal construir un sistema completamente funcional de acuerdo a los requisitos establecidos, sin embargo, cada una de ellas lo consigue de distintas maneras: algunas siguen una serie de etapas sucesivas hasta su fase final, y otras, por el contrario, utilizan una serie de iteraciones en donde cada iteración contiene una serie de etapas distintas.

2. Los modelos de desarrollo de *software* mostrados en este trabajo, los podemos agrupar en dos áreas, de acuerdo al comportamiento de cada uno de ellos, que son modelos tradicionales o secuenciales y modelos evolutivos o iterativos.

3. Los modelos tradicionales enfocan su atención en el progreso a través de una serie de etapas que al final nos conducen al sistema completo. Dentro de esta clasificación están el modelo en cascada, modelo estructurado moderno, modelo en V.

4. Algunos de los problemas encontrados en los modelos tradicionales son los siguientes:
   - Los proyectos raramente siguen el flujo secuencial que impone el modelo, debido a que para el cliente es difícil especificar los requerimientos explícitamente.
   - Una versión funcional del sistema no estará disponible hasta terminar el tiempo de desarrollo estimado, por lo que la tasa de errores es relativamente alta en comparación con los modelos evolutivos.

- No tiene contemplado dentro de su metodología una gestión de riesgos, por lo que se asume que nada afectará el desarrollo del proyecto, cosa que rara vez sucede.

5. Los modelos evolutivos enfocan su atención en el progreso, pero a través de una serie de iteraciones, en donde cada iteración contiene el conjunto de etapas de los modelos tradicionales. En cada iteración se afinan detalles o posibles errores y la iteración final da como resultado el sistema completo. Dentro de esta clasificación están los siguientes modelos: prototipos, espiral, iterativo incremental, programación extrema y ensamblaje de componentes.

6. Algunos de los problemas encontrados en los modelos evolutivos son los siguientes:
   - El análisis de riesgo no es una tarea fácil, por lo tanto se requiere de personal con mucha experiencia en proyectos de *software* para realizar un análisis pertinente.
   - La participación del cliente es indispensable en todo momento, lo cual en algunos casos es difícil de conseguir.
   - El proceso de desarrollo es lento; lo cual confunde al cliente porque piensa que el prototipo que evalúa es el sistema final, cuando en realidad se trata de una versión preliminar.

7. Algunas de las ventajas que proporcionan los modelos evolutivos en comparación con los modelos tradicionales son las siguientes:
   - Evita los problemas de los modelos tradicionales a través de una gestión de riesgos, lo cual nos prepara ante posibles fallas que pueden afectar el desarrollo del sistema si llegaran a ocurrir.

- Elimina los errores a una temprana edad del tiempo de desarrollo, lo que garantiza que el sistema final será completamente funcional de a cuerdo a las especificaciones hechas por el cliente.
- Proporciona mecanismos que garantizan la calidad del *software*.

8. Independientemente de la metodología de desarrollo que se utilice, del área de aplicación, la complejidad del proyecto, el proceso de desarrollo de *software* contiene siempre una serie de fases genéricas, existentes en todos las metodologías. Estas fases son la definición, el desarrollo y el mantenimiento.

9. La elección de una metodología de desarrollo adecuada debe ser llevada a cabo en base a factores claramente establecidos desde el inicio del proyecto, por ejemplo los recursos con los que se cuenta, la complejidad o tamaño del proyecto, el tiempo de desarrollo, ya que si no se establecen adecuadamente existe un alto riesgo de no llevar a cabo el proyecto con éxito.

RECOMENDACIONES

1. Para cualquier metodología de desarrollo de *software*, la complejidad del proyecto puede estimarse de distintas maneras, ya que cada una de ellas está orientada a un tipo de sistema en particular; por ejemplo, los puntos de función se utilizan en aquellos sistemas en donde el usuario tiene un alto grado de interacción y, por el contrario, la métrica de líneas de código se utiliza para sistemas de tiempo real y empotrados en donde la velocidad de respuesta del sistema es un factor crítico.

2. Para los sistemas en donde la complejidad del proyecto sea baja, la innovación tecnológica es un factor irrelevante, y para que se cuente con los recursos necesarios desde un principio los modelos tradicionales serán la mejor opción de desarrollo, ya que una metodología evolutiva le proporcionara más desventajas que ventajas.

3. Para los sistemas en donde la complejidad o tamaño del proyecto sea media o alta, la comunicación con el cliente es constante, la innovación tecnológica es un factor indispensable, la seguridad es un factor crítico y es necesario contar con la información en cualquier momento y se disponga de suficientes recursos principalmente económicos. Definitivamente una metodología evolutiva es la adecuada, porque los cambios no serán significativos y su impacto en el tiempo y costo del sistema no será muy elevado.

4. La gestión de riesgos es una actividad necesaria e indispensable en el desarrollo de un sistema, que se puede incorporar de una manera adecuada a los modelos tradicionales y su correcta utilización a dará como resultado el cumplimiento en la fecha de entrega del sistema final.

5. En cualquier metodología de desarrollo, si por cualquier motivo la planificación del proyecto se atrasa en sus fechas estimadas, una manera de recuperar el tiempo perdido es agregar personal y asignar tareas de todas aquellas actividades que pueden llevarse a cabo en paralelo con las actividades actuales, de otra manera, en vez de beneficiar al proyecto lo perjudicarán.

6. Una manera de ahorrar costos económicos y de tiempo a corto plazo es por medio de la reutilización de *software*, sin embargo, para llevarla a cabo es necesario contar previamente con una biblioteca de componentes. Para un ahorro a largo plazo es conveniente construir dicha biblioteca por medio de proyectos desarrollados con metodologías iterativas, específicamente modelo iterativo incremental y programación extrema, respectivamente.

BIBLIOGRAFÍA

1. Joseph Schmuller. Tr. A. David Garza Marín. **Aprendiendo UML en 24 Horas.** 1º. Edición en español. México: Pearson Educación de México, S.A. de C.V.

2. Kendall, Kenneth E. y Julie E. Tr. Héctor López Hernández. **Análisis y diseño de sistemas.** 1º. Edición en español. México: Prentice - Hall Hispanoamericana S. A.

3. Kendall, Kenneth E. y Julie E. Tr. Ing. Sergio María Ruiz Faudon. **Análisis y diseño de sistemas.** 3º. Edición. México: Prentice - Hall Hispanoamericana S. A.

4. Roger S. Presuman. Tr. Rafael Ojeda Martín, Joaquín Sánchez, Virgilio Galaup, Julio Zurdo Chávez. **Ingeniería del *software.*** 4º. Edición. España: McGraw Hill - Interamericana S. A.

5. **Análisis de requerimientos.** http://trevinca.ei.uvigo.es/~jcmoreno/is/requerimientos.html. (28/02/2003)

6. **Análisis de riesgos.** http://150.214.108.62/~ISP7/analisis-de-riesgos.html. (28/08/2002)

7. **Bienvenidos a *itera e-development process.*** http://www.itera.com.mx/itera/cursos/rup.asp. (28/02/2003)

8. **Breve historia de la calidad.** http://www.ambientando.com/nts/ges_cal/cuerpo.htm. (05/09/2002)

9. **Calidad del *software.*** http://www.multired.com/ciencia/anheresp/calidaddelsoftware.htm. (20/08/2002)

10. **Categorización en tarjetas.** http://www.sidar.org/visitable/tecnicas/Tarj.htm. (15/10/2002)

11. **Ciclo de vida del desarrollo.** http://argos.usb.edu.co/usb-ingsoftware/Documentos/CicloVida/CiclodeVida.html. (12/08/2002)

12. **Coloquios en tecnología de *software.*** http://www.memi.umss.edu.bo/~coloquios/ts/cuatro.html. (28/08/2002)

13. **Cuestionario.** http://www.multired.com/ciencia/anheresp/cuestionario.htm. (20/08/2002)

14. **Curso de gestión de proyectos de desarrollo de *software*.** http://www.inf.udec.cl/~ingsoft/gpds.html. (11/07/2003)

15. **Desarrollo orientado a objetos con UML**. http://www.inf.uach.cl/rvega/asignaturas/info265/doo_uml.pdf. (28/02/2003)

16. **Diagramas de afinidad.** http://www.sidar.org/visitable/tecnicas/Diag.htm. (15/10/2002)

17. **Diseño del sistema.** http://trevinca.ei.uvigo.es/~jcmoreno/is/dis_sistema.html. (12/08/2002)

18. **Diseño detallado.** http://trevinca.ei.uvigo.es/~jcmoreno/is/dis_detallado.html. (12/08/2002)

19. **Dr. Dobb's Sourcebook.** http://www.ddj.com. (28/02/2003)

20. **El ciclo de vida.** http://www.lafacu.com/apuntes/informatica/inge_soft/isw2/default.htm. (11/07/2003)

21. **El ciclo de vida - Fases, tipos de modelos, objetivos de cada fase...** http://www.getec.etsit.upm.es/docencia/gproyectos/planificacion/cvida.htm. (15/10/2002)

22. **El clásico wiki de Ward Cunningham.** http://c2.com/cgi/wiki?ExtremeProgramming. (14/08/2002)

23. **El FAQ de John Brewer.** http://www.jera.com/techinfo/xpfaq.html. (28/08/2002)

24. **El grupo en Yahoo de programación extrema.** http://groups.yahoo.com/group/extremeprogramming. (14/08/2002)

25. **El modelo RAD.** http://www.ldc.usb.ve/~vtheok/cursos/ci3711/apuntes/99-01-14/Info/Modelo%20RAD.htm. (17/10/2002)

26. **El sitio de Josh Kerievsky.** http://www.industriallogic.com/xp/index.html. (14/08/2002)

27. **El sitio de Ron Jeffries.** http://www.xprogramming.com. (14/08/2002)

28. **Elementos de la metodología XP.** http://www.w3aeiou.com/eficiencia/xp3.htm. (28/08/2002)

29. **Entrevista.** http://www.multired.com/ciencia/anheresp/entrevista.htm. (20/08/2002)

30. ***Escolhendo um* modelo de ciclo de vida.** http://www.psphome.hpg.ig.com.br/downloads/Ciclos_de_vida_soft_Slides_Univali.pdf. (28/02/2003)

31. **Espiral.** http://www.itmorelia.edu.mx/Prototipo/INICIO/UNIDAD_2/GRAFICA_ESPIRAL.htm. (20/08/2002)

32. **Estimación de costos.** http://www.itcdguzman.edu.mx/ingsoft/estimac.htm. (10/07/2003)

33. **Estudio de factibilidad.** http://www.itcdguzman.edu.mx/ingsoft/viabili.htm. (11/07/2003)

34. **Etapas de un proyecto - Fases, estructura, etapas...** http://www.getec.etsit.upm.es/docencia/gproyectos/planificacion/etapas.htm. (15/10/2002)

35. **Evaluación de usabilidad formativa.** http://www.sidar.org/visitable/Maner/EvFormat.htm. (17/10/2002)

36. **Exploraciones (Will Wake).** http://xp123.com. (14/08/2002)

37. **Factibilidad del proyecto.** http://www.multired.com/ciencia/anheresp/factibilidaddelproyecto.htm. (20/08/2002)

38. **Factores en el costo del *software.*** http://www.itcdguzman.edu.mx/ingsoft/factorc.htm. (10/07/2003)

39. **Fase del análisis.** http://www.multired.com/ciencia/anheresp/fasedelanalisis.htm. (20/08/2002)

40. **Fases del modelo en espiral.** http://www-gris.det.uvigo.es/~jose/doctorado/proceso_sw/tsld036.htm. (12/08/2002)

41. **Identificación de actividades.** http://www.getec.etsit.upm.es/docencia/gproyectos/planificacion/actividades.htm . (15/10/2002)

42. **Implementación y pruebas.** http://trevinca.ei.uvigo.es/~jcmoreno/is/implementacion.html. (12/08/2002)

43. **Ingeniería del *software*.** http://www.mcc.unam.mx/~cursos/Objetos/clases3_4.html. (19/08/2002)

44. **Ingeniería del *software*.** http://www.umsanet.edu.bo/docentes/gchoque/MAT426TEX.htm. (18/10/2002)

45. **Ingeniería del *software* (etapas).** http://www.euskalnet.net/javierml/softeng/ciclo.htm. (19/08/2002)

46. **Introducción a la ingeniería del *software*.** http://www.ldc.usb.ve/~teruel/ci3715/clases/introSE2.html. (27/08/2002)

47. **ISO.** http://www.geocities.com/gehg48/ISO.html. (23/08/2002)

48. **ISO 9000.** http://www.juanval.net/iso_9000.htm. (05/09/2002)

49. **ISO 9000 aplicada al *software*.** http://www.cp.com.uy/43/iso43.htm. (09/08/2002)

50. **ISO 9000 para desenvolvedor de *software*.** http://c3.eps.ufsc.br/dis_qs99/iso/textoISO.htm. (05/09/2002)

51. **ISO/9000 para el *software*.** http://www.venesoft.com/200004/calidad.htm (23/08/2002)

52. **JOOP: *The Journal of Object-Oriented Programming*.** http://www.sigs.com. (28/02/2003)

53. **La facilidad de uso en el diseño de software.** http://www.microsoft.com/spanish/msdn/articulos/archivo/090201/voices/uidesign.asp. (10/07/2003)

54. **Líneas concurrentes de actividades en el modelo en espiral.** http://www.tid.es/presencia/publicaciones/comsid/esp/articulos/vol51/iptes/images/grandes/fig3.html. (14/08/2002)

55. **Metodología de desarrollo de software orientado por objetos.** http://trevinca.ei.uvigo.es/~jcmoreno/is/proceso.html. (12/08/2002)

56. **Modelo de ciclo de vida de desarrollo de sistemas.** http://www.multired.com/ciencia/anheresp/modelodeciclodevidadedesar.htm. (20/08/2002)

57. **Modelo en espiral.** http://www-gris.det.uvigo.es/~jose/doctorado/proceso_sw/tsld034.htm. (14/08/2002)

58. **Modelos para el desarrollo de *software*.** http://docentes.usaca.edu.co/wildiaz/INGSOF_MODELOS.html. (07/10/2002)

59. **Modelos y técnicas de estimación**. http://www.itcdguzman.edu.mx/ingsoft/modelote.htm. (10/07/2003)

60. **Novática 156: En resumen (¿eXtremismo? Sí, gracias).** http://www.ati.es/novatica/2002/156/enre156.html. (12/08/2002)

61. **Observación directa.** http://www.multired.com/ciencia/anheresp/observaciondirecta.htm. (20/08/2002)

62. **Programación extrema y *software* libre.** http://congreso.hispalinux.es/ponencias/ferrer/robles-ferrer-ponencia-hispalinux-2002.html. (27/03/2003)

63. ***Project Management Institute.*** http://www.pmi.org. (28/02/2003)

64. **Prototipado.** http://www.sidar.org/visitable/Maner/Prototipado.htm. (07/10/2002)

65. **Prototipado de alta fidelidad.** http://www.sidar.org/visitable/tecnicas/High.htm. (07/10/2002)

66. **Prototipado de baja fidelidad.** http://www.sidar.org/visitable/tecnicas/Low.htm. (07/10/2002)

67. **Prototipado de papel.** http://www.sidar.org/visitable/nuevos/Papel.htm. (07/10/2002)

68. **Prototipado horizontal.** http://www.sidar.org/visitable/tecnicas/Horiz.htm. (07/10/2002)

69. **Prototipado rápido.** http://www.sidar.org/visitable/nuevos/Rapido.htm. (07/10/2002)

70. **Prototipado vertical.** http://www.sidar.org/visitable/tecnicas/Vert.htm. (07/10/2002)

71. **Prototipado y categorización.** http://www.sidar.org/visitable/prototype.htm. (07/10/2002)

72. **Pruebas de programas.** http://www.upv.es/protel/usr/jotrofer/pascal/testing.htm. (04/09/2002)

73. **RAD (*Rapid Application Development*) *and* JAD (*Joint Application Design*) *Workshops*.** http://www.ucc.ie/ucc/research/hfrg/projects/respect/urmethods/rad.htm. (15/10/2002)

74. ***RAD Qualité & Ingénierie REENGINEERING développement d'applications.*** http://www.rad.fr. (18/10/2002)

75. **Recursos.** http://www.itcdguzman.edu.mx/ingsoft/recurso.htm. (10/07/2003)

76. **Revista Informática: informática.CL.** http://www.informatica.cl/marzo2002/tejado.htm. (28/08/2002)

77. ***Software Development Magazine.*** http://www.sdmagazine.com. (28/02/2003)

78. **Tendencias tecnológicas y del mercado.** http://www.map.es/csi/silice/Dsamed21.html. (20/08/2002)

79. ***The Seven Management and Planning Tools.*** http://www.goalqpc.com/whatweteach/RESEARCH/7mp.html. (15/10/2002)

80. ***The Seven Quality Control Tools.*** http://www.goalqpc.com/whatweteach/RESEARCH/7qc.html. (15/10/2002)

81. **Una gentil introducción, de Don Wells.** http://www.extremeprogramming.org. (14/08/2002)

82. **UNIVALI *Universidade do Vale do Itajaí.*** http://www.sj.univali.br/~fleury/es.html. (28/02/2003)

83. **VB RAD *Home Page*.** http://www.vbrad.com. (18/10/2002)

84. **Windows TI Magazine 63 - Abril 2002 - El observador - Programación eXtrema.**
http://www.windowstimag.com/atrasados/2002/63_abr02/articulos/observa_extrema.asp. (14/08/2002)

85. **XP: *Extreme Programming* (Programación Extrema).**
http://www.w3aeiou.com/eficiencia/xp1.htm. (14/08/2002)