

# TEMA 2:

## EL PROCESO DE DESARROLLO DEL SOFTWARE

### ÍNDICE

1.-	PARTICIPANTES .....	1
1.1.-	USUARIOS .....	1
1.2.-	ADMINISTRACIÓN .....	2
1.3.-	AUDITORES .....	3
1.4.-	ANALISTA DE SISTEMAS .....	3
1.5.-	DISEÑADORES DE SISTEMAS.....	3
1.6.-	PROGRAMADORES.....	4
1.7.-	PERSONAL DE OPERACIONES .....	4
2.-	EL CICLO DE VIDA DEL SOFTWARE.....	4
2.1.-	VISIÓN GENÉRICA DE LA INGENIERÍA DEL SOFTWARE.....	4
2.2.-	CONCEPTO DE CICLO DE VIDA.....	8
2.3.-	CICLO DE VIDA EN CASCADA.....	8
2.4.-	EL MODELO INCREMENTAL. ....	12
2.5.-	USO DE TÉCNICAS DE CUARTA GENERACIÓN.....	13
2.6.-	CONSTRUCCIÓN DE PROTOTIPOS. ....	15
2.7.-	EL MODELO EN ESPIRAL.....	18
2.8.-	INGENIERÍA INVERSA Y REINGENIERÍA DEL SOFTWARE.....	20
3.-	METODOLOGÍAS DE DESARROLLO DE SOFTWARE.....	21
3.1.-	INTRODUCCIÓN .....	21
3.2.-	CARACTERÍSTICAS PRINCIPALES DE LAS METODOLOGÍAS .....	22
3.3.-	CLASIFICACIÓN DE LAS METODOLOGÍAS .....	23
3.4.-	PRINCIPALES METODOLOGÍAS DE DESARROLLO .....	26



## 1.- PARTICIPANTES

La comunicación entre los diferentes participantes será difícil debido al empleo de distintos lenguajes. Por tanto es importante saber que puede esperar el analista de los usuarios y viceversa.

Participantes:

- Usuarios
- Administración
- Auditores
- Analistas del sistema
- Diseñadores del sistema
- Programadores
- Personal de operaciones

### 1.1.- USUARIOS

---

Para quienes se construye el sistema. Con ellos se tendrán las entrevistas en detalle para efectuar el análisis.

Se ha de procurar estar en contacto directo con el usuario, pero si esto no es posible, entonces, la documentación generada por el analista se vuelve más importante.

Las herramientas de modelado que se utilizan para describir el sistema de manera formal y rigurosa son esenciales para evitar malos entendidos.

**Usuarios operacionales:** Están en contacto diario con el sistema.

- Más preocupados por los detalles de funcionamiento y la interfaz. *Ej.: localización de la información en la pantalla y en los informes, tamaño de la letra avisos en caso de error.*
- Tienen una visión del panorama local del sistema, conocen los detalles. Poca visión (conocimiento) del panorama global.

Por tanto el analista debe desarrollar modelos que permitan ambos panoramas.

- Piensan en los sistemas en términos físicos. No entienden de *funciones* o *tipos de datos*.

**Supervisores:** *jefe de turno, gerente, ejecutivo...* con un grupo de usuarios operacionales a su cargo y son responsables de sus logros.

- Posiblemente hayan sido promovidos desde cargos en el nivel inferior, están por tanto familiarizados con el trabajo de los usuarios operacionales (no siempre).
- Su visión puede ser también local, pero ha olvidado algunos detalles.
- El sistema le permitirá una mejor supervisión. Tiene pues una perspectiva distinta a la de los usuarios operacionales. Ve el sistema como un modo de disminuir el número de usuarios operacionales, esto originará batallas políticas.
- Es el contacto cotidiano primario con el analista. Definirá los requerimientos y la política de la empresa.

**Usuario ejecutivo:** No se involucra directamente

- Puede proporcionar la iniciativa o ser sólo la autoridad que financie la solicitud.

- Probablemente no fue nunca usuario operativo y si lo fue, se le ha olvidado. Por tanto, no está en situación de definir los requerimientos del sistema para aquellos que lo están utilizando diariamente. (Excepto si se trata de un sistema DSS)
- Se interesan más por el panorama global que por el detalle. Necesitan pues herramientas de modelación que permitan dar un panorama global.

### ***Resumen de características***

<b>Usuario operacional</b>	<b>Usuario supervisor</b>	<b>Usuario ejecutivo</b>
Usualmente tienen un panorama local.	Pueden o no tener un panorama local	Tienen un panorama global
Hacen funcionar el sistema	Está familiarizado con la operación pero ha olvidado los detalles	Provee la iniciativa para el proyecto
Tiene una visión física del sistema	Lo rigen consideraciones presupuestarias.  Actúa a menudo como intermediario entre los usuarios y los niveles superiores de administración.	No tiene experiencia operacional directa.  Tiene preocupaciones estratégicas.

## **1.2.- ADMINISTRACIÓN**

### **1.2.1.- ADMINISTRADORES DE INFORMÁTICA.**

Encargados del proyecto de sistemas. Los de nivel superior se encargan de la administración global y de la distribución de recursos de todo el personal técnico de la organización de creación o desarrollo de sistemas.

### **1.2.2.- ADMINISTRACIÓN GENERAL.**

Administradores de nivel superior, no están directamente involucrados con la organización de informática ni son de la organización usuaria.

La principal interacción entre el analista y la administración tiene que ver con los recursos que se asignarán al proyecto.

El analista finalmente redactará un documento que diga: *"El nuevo sistema deberá llevar a cabo las funciones X, Y y Z, y deberá desarrollarse en seis meses, con no más de N programadores y con un costo máximo de QQQQQQ pts."*

La administración querrá que se le asegure que el proyecto de desarrollo del sistema se está manteniendo dentro de estos márgenes. Esto es asunto de la administración de proyectos, no del análisis del sistema (aunque en ocasiones coincidan).

### **1.3.- AUDITORES**

---

Según sea el tamaño del proyecto y la naturaleza de la organización, pudiera haber auditores, personal de control de calidad o miembro del departamento de normas o estándares participando en el proyecto. El objetivo general de este equipo es asegurar que el sistema se desarrolle de acuerdo con diversos estándares o normas externos.

Problemas:

- A menudo no se involucran hasta el final del proyecto, cuando es más difícil hacer cambios importantes en el sistema.
- No están familiarizados con la notación para documentar requerimientos. Es importante asegurarse de que los modelos del sistema sean comprensibles.
- Se interesan más por las formas que por el contenido. Si los documentos no tienen la presentación exacta que se exige pudieran ser rechazados.

### **1.4.- ANALISTA DE SISTEMAS**

---

**Funciones:**

- Describir y documentar detalles de la política de un negocio que pudieran existir como "tradiciones".
- Distinguir los problemas existentes y sus causas. Ayudar al usuario a explorar aplicaciones novedosas y más útiles de las computadoras así como formas nuevas de hacer negocios.
- Mediar entre los distintos participantes y obtener un consenso.

**Cualidades necesarias:**

- Habilidad en el manejo de las personas.
- Conocimientos de la aplicación para entender y apreciar los asuntos de los usuarios.
- Habilidad en computación para entender los usos potenciales del hardware y del software en los asuntos del usuario.
- Mente lógica y organizada: debe ser capaz de ver un sistema desde diferentes perspectivas, debe poder dividirlo en niveles de subsistemas y ser capaz de pensar en términos abstractos además de físicos.

### **1.5.- DISEÑADORES DE SISTEMAS**

---

Su función es transcribir los datos provenientes de análisis en un diseño arquitectónico de alto nivel que servirá de base para el trabajo de los programadores.

La comunicación con el analista es en dos sentidos:

El analista tiene que ofrecer información detallada para que el diseñador pueda elaborar un diseño tecnológicamente superior.

El diseñador debe proveer suficiente información para que el analista pueda darse cuenta de si los requerimientos del usuario son tecnológicamente posibles.

## 1.6.- PROGRAMADORES.

---

Reciben el diseño y lo codifican.

Esta división es muy teórica y aunque aparentemente el analista no tenga ningún contacto con el programador, muchas veces no ocurre así.

En ocasiones, la misma persona realiza las tres funciones. Otras veces el analista es además el administrador del proyecto y tendrá contacto con los programadores.

A menudo es el programador el que descubre errores o ambigüedades en los requerimientos y necesita pedir una aclaración al analista.

En algunas organizaciones en lugar de la estructura vertical en el desarrollo de proyectos se tiene una estructura horizontal en la que se le da al personal técnico superior (analista) todas las tareas de alto nivel: análisis, diseño y codificación de módulos de alto nivel. Similarmente a las personas de nivel técnico inferior se les dará tareas detalladas de bajo nivel en las áreas de análisis, de diseño y de programación. En la medida en que esto se siga, los analistas y los programadores mantendrán un contacto cercano durante todo el proyecto.

## 1.7.- PERSONAL DE OPERACIONES

---

Responsable del centro de cómputo, la red de telecomunicaciones, la seguridad del hardware y del software, además de la ejecución de los programas, el montaje de los discos y el manejo de la salida de las impresoras. Todo esto sucede después de haber sido tanto analizado como programado y probado el sistema.

## 2.- EL CICLO DE VIDA DEL SOFTWARE

---

### 2.1.- VISIÓN GENÉRICA DE LA INGENIERÍA DEL SOFTWARE.

---

Ingeniería del Software. Definiciones:

- |  |
|--|
| <ul style="list-style-type: none"><li>• <i>Producción sistemática y mantenimiento de productos software</i></li></ul>  |
| <ul style="list-style-type: none"><li>• <i>La aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software</i></li></ul> |

Se basa en un enfoque de calidad y esto exige unos métodos de trabajo.

La ingeniería abarca:

- **Métodos:** ¿Cómo se ha de construir técnicamente el software?
- **Herramientas:** Elementos que permiten automatizar las tareas.
- **Procedimientos:** Protocolo definido durante el diseño del software: reglas, normas, especificaciones, plazos, controles de calidad.

Independientemente del ciclo de vida que se siga, del área de aplicación, y del tamaño y la complejidad del proyecto, el proceso de desarrollo de software contiene

siempre una serie de fases genéricas, existentes en todos los paradigmas. Estas fases son la definición, el desarrollo y el mantenimiento.

### 2.1.1.- DEFINICIÓN.

La fase de definición se centra en el **QUE**. Durante esta fase, se intenta identificar:

- qué **información** es la que tiene que ser procesada.
- qué **función y rendimiento** son los que se esperan.
- qué **restricciones de diseño** existen.
- qué **interfaces** deben utilizarse.
- qué **lenguaje de programación, sistema operativo y soporte hardware** van a ser utilizados.
- qué **criterios de validación** se necesitan para conseguir que el sistema final sea correcto.

Aunque los pasos concretos dependen del modelo de ciclo de vida utilizado, en general en esta fase se realizarán tres tareas específicas:

⇒ *Análisis del sistema.*

Como ya hemos visto, el análisis del sistema define el papel de cada elemento de un sistema informático, estableciendo cuál es el papel del software dentro de ese sistema.

⇒ *Análisis de requisitos del software.*

El análisis del sistema proporciona el ámbito del software, su relación con el resto de componentes del sistema, pero antes de empezar a desarrollar es necesario hacer una definición más detallada de la función del software.

Existen dos formas de realizar el análisis y refinamiento de los requisitos del software:

- Por una parte, se puede hacer un análisis formal del ámbito de la información para establecer modelos del flujo y la estructura de la información. Luego se amplían unos modelos para convertirlos en una especificación del software.
- La otra alternativa consiste en construir un prototipo del software, que será evaluado por el cliente para intentar consolidar los requisitos. Los requisitos de rendimiento y las limitaciones de recursos se traducen en directivas para la fase de diseño.

El análisis y definición de los requisitos es una tarea que debe llevarse a cabo conjuntamente por el desarrollador de software y por el cliente. La especificación de requisitos del software es el documento que se produce como resultado de esta etapa.

⇒ *Planificación del proyecto software.*

Durante esta etapa se lleva a cabo el análisis de riesgos, se definen los recursos necesarios para desarrollar el software y se establecen las estimaciones de tiempo y costes. El propósito de esta etapa de planificación es proporcionar una indicación preliminar de la viabilidad del proyecto de acuerdo con el coste y con la agenda que se

hayan establecido. Posteriormente, la gestión del proyecto durante el desarrollo del mismo realiza y revisa el plan de proyecto de software.

### 2.1.2.- DESARROLLO.

La fase de definición se centra en el **CÓMO**.

- cómo ha de ser la **arquitectura de la aplicación**.
- cómo han de ser las **estructuras de datos**.
- cómo han de implementarse los **detalles procedimentales** de los módulos.
- cómo van a ser las **interfaces**.
- cómo ha de traducirse el diseño a un lenguaje de programación (**codificación**).
- cómo van a realizarse las **pruebas**.

Aunque, al igual que antes, los pasos concretos dependen del modelo de ciclo de vida utilizado, en general se realizarán cuatro tareas específicas:

#### ⇒ ***Diseño.***

El diseño del software traduce los requisitos a un conjunto de representaciones (gráficas, en forma de tabla o basadas en algún lenguaje apropiado) que describen cómo van a estructurarse los datos, cuál va a ser la arquitectura de la aplicación, cuál va a ser la estructura de cada programa y cómo van a ser las interfaces. Es necesario seguir criterios de diseño que nos permitan asegurar la calidad del producto.

Una vez finalizado el diseño es necesario revisarlo para asegurar la completitud y el cumplimiento de los requisitos del software.

#### ⇒ ***Codificación.***

En esta fase, el diseño se traduce a un lenguaje de programación, dando como resultado un programa ejecutable. La buena calidad de los programas desarrollados depende en gran medida de la calidad del diseño.

Una vez codificados los programas debe revisarse su estilo y claridad, y se comprueba que haya una correspondencia con la estructura de los mismos definida en la fase de diseño.

El listado fuente de cada módulo (o el programa fuente en soporte magnético) pasa a formar parte de la configuración del sistema.

#### ⇒ ***Pruebas.***

Una vez que tenemos implementado el software es preciso probarlo, para detectar errores de codificación, de diseño o de especificación. Las pruebas son necesarias para encontrar el mayor número posible de errores antes de entregar el programa al cliente.

Es necesario probar cada uno de los componentes por separado (cada uno de los módulos o programas) para comprobar el rendimiento funcional de cada una de estas unidades.

A continuación se procede a integrar los componentes para probar toda la arquitectura del software, y probar su funcionamiento y las interfaces. En este punto hay que comprobar si se cumplen todos los requisitos de la especificación.



Se puede desarrollar un plan y procedimiento de pruebas y guardar información sobre los casos de pruebas y los resultados de las mismas.

### 2.1.3.- MANTENIMIENTO.

La fase de mantenimiento se centra en los cambios que va a sufrir el software a lo largo de su vida útil. Como ya hemos dicho, estos cambios pueden deberse a la corrección de errores, a cambios en el entorno inmediato del software o a cambios en los requisitos del cliente, dirigidos normalmente a ampliar el sistema.

La fase de mantenimiento vuelve a aplicar los pasos de las fases de definición y de desarrollo, pero en el contexto de un software ya existente y en funcionamiento.

### **Visión genérica de la Ingeniería del Software.**

• <u>DEFINICIÓN.</u>	<u>¿QUÉ?</u>
	<p>Análisis del sistema.</p> <p><i>Establecer el ámbito del software.</i></p> <p>Análisis de requisitos del sistema software.</p> <p><i>Definición detallada de la función del software.</i></p> <p>Planificación.</p> <p><i>Análisis de riesgos.</i></p> <p><i>Asignación de recursos.</i></p> <p><i>Definición de tareas.</i></p> <p><i>Estimación de costes.</i></p>
• <u>DESARROLLO.</u>	<u>¿CÓMO?</u>
	<p>Diseño.</p> <p><i>Arquitectura de la aplicación.</i></p> <p><i>Estructura de los datos.</i></p> <p><i>Estructura interna de los programas.</i></p> <p><i>Diseño de las interfaces.</i></p> <p>Codificación.</p> <p>Pruebas.</p>
• <u>MANTENIMIENTO.</u>	<u>EL CAMBIO.</u>
	<p><i>Corrección de errores.</i></p> <p><i>Cambios en el entorno.</i></p> <p><i>Cambios en los requisitos.</i></p>

## 2.2.- CONCEPTO DE CICLO DE VIDA.

---

Los sistemas informáticos se desarrollan en una serie de pasos. Esta secuencia de pasos se conoce como **ciclo de vida del software**.

Por **ciclo de vida**, se entiende la sucesión de etapas por las que pasa el software desde que un nuevo proyecto es concebido hasta que se deja de usar.

Cada una de estas etapas lleva asociada una serie de tareas que deben realizarse, y una serie de documentos (en sentido amplio: software) que serán la salida de cada una de estas fases y servirán de entrada en la fase siguiente.

Existen diversos modelos de ciclo de vida, es decir, diversas formas de ver el proceso de desarrollo de software. En esta unidad veremos algunos de los principales modelos de ciclo de vida.

La elección de uno u otro se realiza de acuerdo con la naturaleza del proyecto y de la aplicación, los métodos a usar y los controles y entregas requeridos.

¿Para que sirve seguir un modelo de ciclo de vida?

- Definir las actividades a llevarse a cabo en un proyecto de desarrollo de sistemas.
- Lograr congruencia entre la multitud de proyectos de desarrollo de sistemas en una misma organización.
- Proporcionar puntos de control y revisión administrativos de las decisiones sobre continuar o no el proyecto y para controlar si se están cumpliendo los objetivos o si hace falta más recursos.

## 2.3.- CICLO DE VIDA EN CASCADA.

---

Este paradigma es el más antiguo de los empleados en la IS (Ingeniería del Software) y se desarrolló a partir del ciclo convencional de una ingeniería. No hay que olvidar que la IS surgió como copia de otras ingenierías, especialmente de las del hardware, para dar solución a los problemas más comunes que aparecían al desarrollar sistemas de software complejos.

Es un ciclo de vida en sentido amplio, que incluye no sólo las etapas de ingeniería sino toda la vida del producto: las pruebas, el uso (la vida útil del software) y el mantenimiento, hasta que llega el momento de sustituirlo (Figura 1)

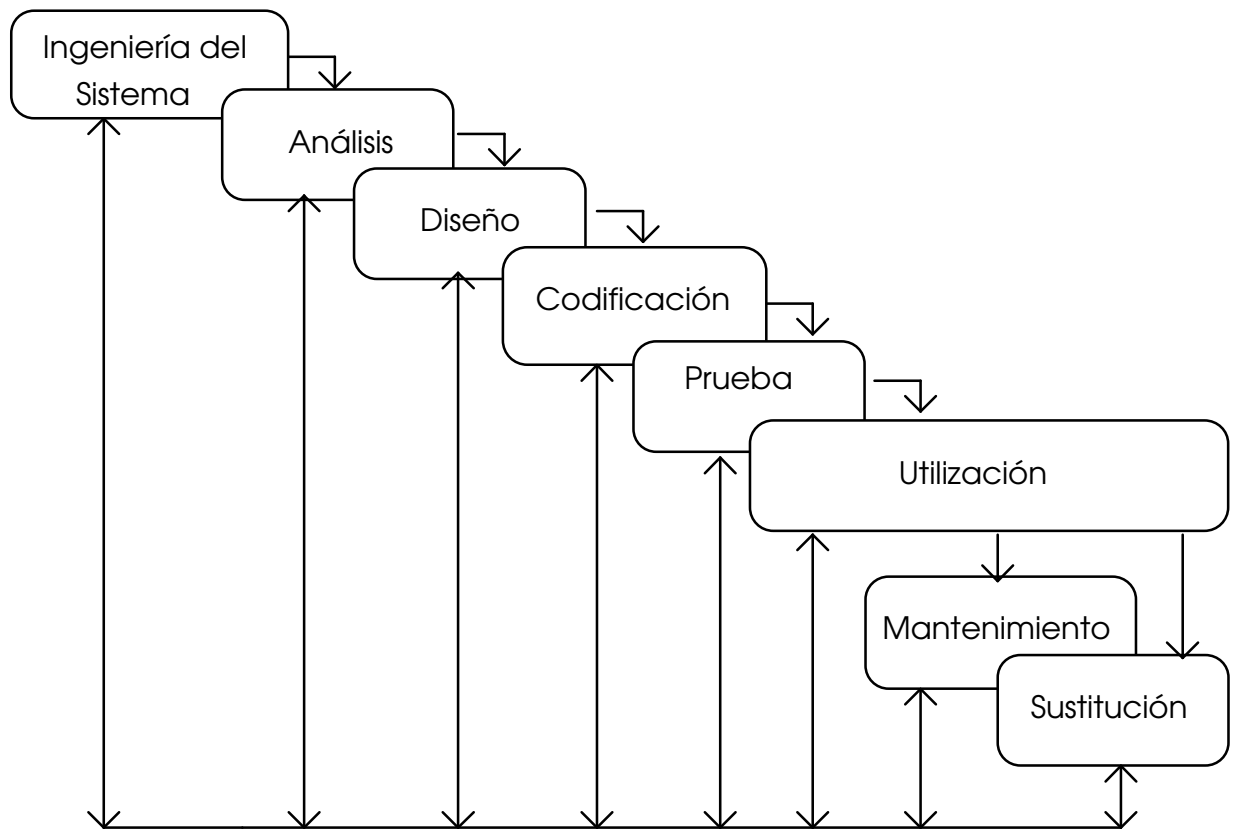


Figura 1. Ciclo de vida en cascada.

El ciclo de vida en cascada exige un enfoque sistemático y secuencial del desarrollo de software, que comienza en el nivel de la ingeniería de sistemas y avanza a través de fases secuenciales sucesivas. Estas fases son las siguientes:

### 2.3.1.- INGENIERÍA Y ANÁLISIS DEL SISTEMA.

El software es siempre parte de un sistema mayor, por lo que siempre va a interrelacionarse con otros elementos, ya sea hardware, máquinas o personas. Por esto, el primer paso del ciclo de vida de un proyecto consiste en un análisis de las características y el comportamiento del sistema del cual el software va a formar parte.

La ingeniería del sistema comprende, por tanto, los requisitos globales a nivel del sistema, así como una cierta cantidad de análisis y de diseño a nivel superior, es decir sin entrar en mucho detalle.

### 2.3.2.- ANÁLISIS DE REQUISITOS DEL SOFTWARE.

El análisis de requisitos debe ser más detallado para aquellos componentes del sistema que vamos a implementar mediante software. El ingeniero del software debe comprender cuáles son los datos que se van a manejar, cuál va a ser la función que tiene que cumplir el software, cuáles son los interfaces requeridos y cuál es el rendimiento que se espera lograr.

Los requisitos, tanto del sistema como del software deben documentarse y revisarse con el cliente.

### 2.3.3.- DISEÑO.

El diseño es el proceso que traduce los requisitos en una representación del software de forma que pueda conocerse la estructura de los datos, la arquitectura de las aplicaciones, la estructura interna de los programas y las interfaces antes de comenzar la codificación.

Al igual que el análisis, el diseño debe documentarse y forma parte de la configuración del software (el control de configuraciones es lo que nos permite realizar cambios en el software de forma controlada y no traumática para el cliente).

### 2.3.4.- CODIFICACIÓN.

La codificación consiste en la traducción del diseño a un formato que sea legible para la máquina. Si el diseño es lo suficientemente detallado, la codificación es relativamente sencilla, y puede hacerse - al menos en parte - de forma automática, usando generadores de código.

### 2.3.5.- PRUEBA.

Una vez que ya tenemos el programa ejecutable, comienza la fase de pruebas. El objetivo es comprobar que no se hayan producido errores en alguna de las fases de traducción anteriores, especialmente en la codificación. Para ello deben probarse todas las sentencias, no sólo los casos normales y todos los módulos que forman parte del sistema.

### 2.3.6.- UTILIZACIÓN.

Una vez superada la fase de pruebas, el software se entrega al cliente y comienza la vida útil del mismo. La fase de utilización se solapa con las posteriores - el mantenimiento y la sustitución - y dura hasta que el software, ya reemplazado por otro, deje de utilizarse.

### 2.3.7.- MANTENIMIENTO.

El software sufrirá cambios a lo largo de su vida útil. Estos cambios pueden ser debidos a tres causas:

- Que, durante la utilización, el cliente detecte errores en el software: los errores latentes.
- Que se produzcan cambios en alguno de los componentes del sistema informático: por ejemplo cambios en la máquina, en el sistema operativo o en los periféricos.
- Que el cliente requiera modificaciones funcionales (normalmente ampliaciones) no contempladas en el proyecto.

En cualquier caso, el mantenimiento supone volver atrás en el ciclo de vida, a las etapas de codificación, diseño o análisis dependiendo de la magnitud del cambio.

El modelo en cascada, a pesar de ser lineal, contiene flujos que permiten la vuelta atrás. Así, desde el mantenimiento se vuelve al análisis, el diseño o la codificación, y también desde cualquier fase se puede volver a la anterior si se detectan fallos. Estas

vueltras atrás no son controladas, ni quedan explícitas en el modelo, y este es uno de los problemas que presenta este paradigma

### 2.3.8.- SUSTITUCIÓN.

La vida del software no es ilimitada y cualquier aplicación, por buena que sea, acaba por ser sustituida por otra más amplia, más rápida o más bonita y fácil de usar.

La sustitución de un software que está funcionando por otro que acaba de ser desarrollado es una tarea que hay que planificar cuidadosamente y que hay que llevar a cabo de forma organizada.

Es conveniente realizarla **por fases**, si esto es posible, no sustituyendo todas las aplicaciones de golpe, puesto que la sustitución conlleva normalmente un aumento de trabajo:

- para los usuarios, que tienen que acostumbrarse a las nuevas aplicaciones
- para los implementadores, que tienen que corregir los errores que aparecen.

Es necesario hacer un **trasvase de la información** que maneja el sistema viejo a la estructura y el formato requeridos por el nuevo.

Además, es conveniente **mantener los dos sistemas funcionando en paralelo** durante algún tiempo para comprobar que el sistema nuevo funcione correctamente y para asegurarnos el funcionamiento normal de la empresa aún en el caso de que el sistema nuevo falle y tenga que volver a alguna de las fases de desarrollo.

La sustitución implica el **desarrollo de programas para la interconexión** de ambos sistemas, el viejo y el nuevo, y para trasvasar la información entre ambos, evitando la duplicación del trabajo de las personas encargadas del proceso de datos, durante el tiempo en que ambos sistemas funcionen en paralelo.

### ***Problemas del ciclo de vida en cascada:***

El uso de la implantación ascendente: Se espera que los programadores lleven a cabo primero sus pruebas modulares, luego las pruebas del subsistema, y finalmente las pruebas del sistema mismo. Los errores se encuentran tarde.

En realidad los proyectos no siguen un ciclo de vida estrictamente secuencial como propone el modelo. Siempre hay iteraciones. El ejemplo más típico es la fase de mantenimiento, que implica siempre volver a alguna de las fases anteriores, pero también es muy frecuente en que una fase, por ejemplo el diseño, se detecten errores que obliguen a volver a la fase anterior, el análisis.

Es difícil que se puedan establecer inicialmente todos los requisitos del sistema. Normalmente los clientes no tienen conocimiento de la importancia de la fase de análisis o bien no han pensado en todo detalle que es lo que quieren que haga el software. Los requisitos se van aclarando y refinando a lo largo de todo el proyecto, según se plantean dudas concretas en el diseño o la codificación. Sin embargo, el ciclo de vida clásico requiere la definición inicial de todos los requisitos y no es fácil acomodar en él las incertidumbres que suelen existir al comienzo de todos los proyectos.

Hasta que se llega a la fase final del desarrollo: la codificación, no se dispone de una versión operativa del programa. Como la mayor parte de los errores se detectan

cuando el cliente puede probar el programa no se detectan hasta el final del proyecto, cuando son más costosos de corregir y más prisa (y más presiones) hay por que el programa se ponga definitivamente en marcha.

Todos estos problemas son reales, pero de todas formas es mucho mejor desarrollar software siguiendo el modelo de ciclo de vida en cascada que hacerlo sin ningún tipo de guías. Además, este modelo describe una serie de pasos genéricos que son aplicables a cualquier otro paradigma, refiriéndose la mayor parte de las críticas que recibe a su carácter secuencial.

## 2.4.- EL MODELO INCREMENTAL.

Este modelo corrige la necesidad de una secuencia no lineal de pasos de desarrollo. El sistema software se va creando añadiendo componentes funcionales al sistema (llamados incrementos). En cada paso sucesivo, se actualiza el sistema con nuevas funcionalidades o requisitos, es decir, cada versión o refinamiento parte de una versión previa y le añade nuevas funciones. El sistema software ya no se ve como una única entidad monolítica con una fecha fija de entrega, sino como una integración de resultados sucesivos obtenidos después de cada iteración.

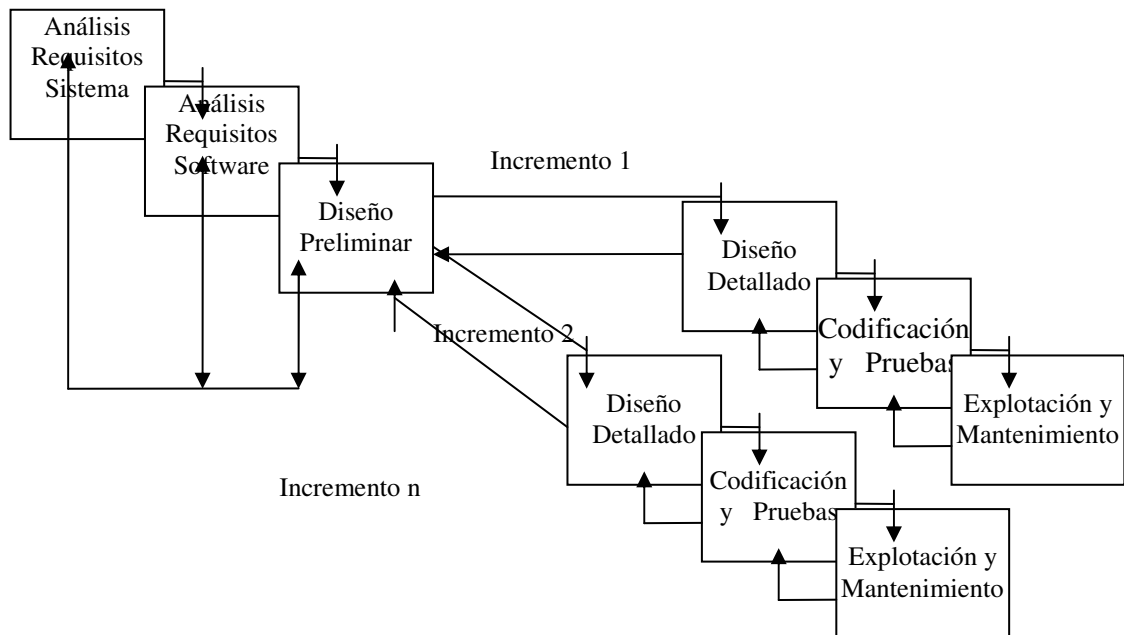


Figura 2. Modelo incremental

El modelo incremental se ajusta a entornos de alta incertidumbre, por no tener la necesidad de poseer un conjunto exhaustivo de requisitos, especificaciones, diseños, etc, al comenzar el sistema, ya que cada refinamiento amplía los requisitos y las especificaciones derivadas de la fase anterior.

(Ej.: *Procesador de texto*)

El modelo incremental constituyó un avance sobre el modelo en cascada, pero también presenta problemas. Aunque permite el cambio continuo de requisitos, aún existe el **problema** de determinar si los requisitos propuestos son válidos. Los errores en los requisitos se detectan tarde y su corrección resulta tan costosa como en el modelo en cascada.

## 2.5.- USO DE TÉCNICAS DE CUARTA GENERACIÓN.

---

Por *técnicas de cuarta generación* se entiende un conjunto muy diverso de métodos y herramientas que tienen por objeto el facilitar el desarrollo de software. Pero todos ellos tienen algo en común: facilitan al que desarrolla el software el especificar algunas características del mismo a alto nivel. Luego, la herramienta genera automáticamente el código fuente a partir de esta especificación.

Los tipos más habituales de generadores de código cubren uno o varios de los siguientes aspectos:

- **Acceso a bases de datos utilizando lenguajes de consulta de alto nivel** (derivados normalmente de SQL). Con ello no es necesario conocer la estructura de los ficheros o tablas ni de sus índices.
- **Generación de código.** A partir de una especificación de los requisitos se genera automáticamente toda la aplicación.
- **Generación de pantallas.** Permiten diseñar la pantalla dibujándola directamente, incluyendo además el control del cursor y la gestión de errores de los datos de entrada.
- **Gestión de entornos gráficos.**
- **Generación de informes.** (de forma similar a las pantallas).

Esta generación automática permite reducir la duración de las fases del ciclo de vida clásico, especialmente la fase de codificación, quedando el ciclo de vida según se indica en la figura 3 con las siguientes fases:

### **Recolección de requisitos:**

Que pueden ser traducidos directamente a código fuente usando un generador de código. Pero el problema es el mismo que se plantea en el ciclo de vida clásico: es muy difícil que se puedan establecer todos los requisitos desde el comienzo: el cliente puede no estar seguro de lo que necesita, o, aunque lo sepa, puede ser difícil expresarlo de la forma en que precisa la herramienta de cuarta generación para poder entenderla.

### **Estrategia de diseño:**

Si la especificación es pequeña, podemos pasar directamente del análisis de requisitos a la generación automática de código, sin realizar ningún tipo de diseño. Pero si la aplicación es grande, se producirán los mismos problemas que si no usamos técnicas de cuarta generación: mala calidad, dificultad de mantenimiento y poca aceptación por parte del cliente). Es necesario, por tanto, realizar un cierto grado de diseño (al menos lo que hemos llamado una estrategia de diseño, puesto que el propio generador se encarga de parte de los detalles del diseño tradicional: descomposición modular, estructura lógica y organización de los ficheros, etc.).

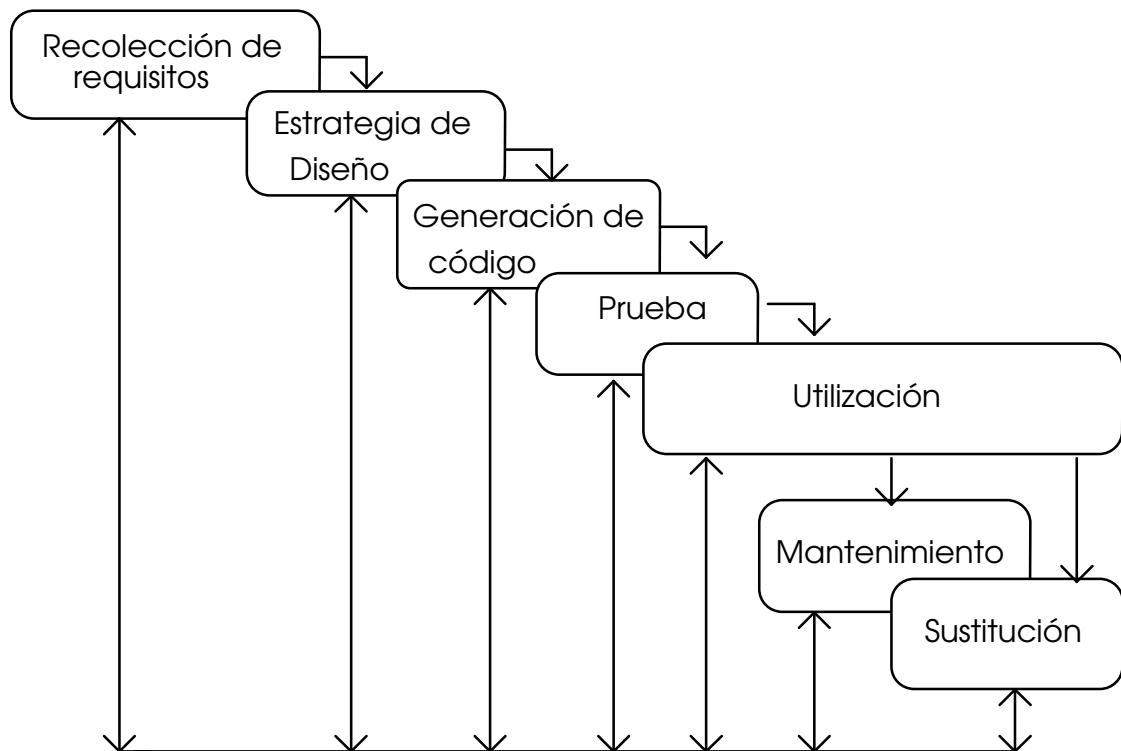


Figura 3. Ciclo de vida usando técnicas de cuarta generación.

**Generación de código:**

Las herramientas de cuarta generación se encargan también de producir automáticamente la documentación del código generado, pero esta documentación es de ordinario muy parca y, por ello, difícil de seguir. Es necesario completarla hasta obtener una documentación con sentido.

**Pruebas:**

Podemos suponer (aunque nunca hay que fiarse) que el código generado es correcto y acorde con la especificación, y que no contiene los típicos errores de la codificación manual. Pero en cualquier caso es necesaria la fase de pruebas, en primer lugar para comprobar la eficiencia del código generado (la generación automática de los accesos a bases puede producir código muy ineficiente cuando el volumen de información es grande (p.ej.: las distintas formas de relacionar tablas en SQL), también para detectar los errores en la especificación a partir de la cual se generó el código, y, por último, para que el cliente compruebe si el producto final satisface sus necesidades.

El resto de las fases del ciclo de vida usando estas técnicas es igual a las del paradigma del ciclo de vida en cascada, del que este no es más que una adaptación a las nuevas herramientas de producción de software.

Estas herramientas consiguen reducir el tiempo de desarrollo de software, eliminando las tareas más repetitivas y tediosas (ej. control de la entrada/salida por terminal) y aumentan la productividad de los programadores, por lo que son ampliamente utilizadas en la actualidad, especialmente si nos referimos a el acceso a bases de datos, la gestión de la entrada/salida por terminal y la generación de informes, y forman parte de muchos de los lenguajes de programación que se usan actualmente, sobre todo en el campo del software de gestión.



No obstante, entre las *críticas* más habituales están:

**No son más fáciles de utilizar que los lenguajes de tercera generación.** En concreto, muchos de los lenguajes de especificación que utilizan pueden considerarse como lenguajes de programación, de un nivel algo más alto que los anteriores, pero que no logran prescindir de la codificación en sí, sino que simplemente la disfrazan de ‘especificación’.

**El código fuente que producen es ineficiente.**(el ejemplo de antes de SQL). Al estar generado automáticamente no pueden hacer uso de los *trucos* habituales para aumentar el rendimiento, que se basan en el buen conocimiento de cada caso particular. Pero la reducción de los tiempos de desarrollo y el continuo aumento de la potencia de cálculo de los ordenadores compensan ampliamente esta menor eficiencia (salvo en excepciones).

## 2.6.- CONSTRUCCIÓN DE PROTOTIPOS.

---

Dos de las críticas que se hacían al modelo de ciclo de vida en cascada eran:

- que es difícil tener claros todos los requisitos del sistema al inicio del proyecto,
- y que no se dispone de una versión operativa del programa hasta las fases finales del desarrollo, lo que dificulta la detección de errores y deja también para el final el descubrimiento de los requisitos inadvertidos en las fases de análisis.

Para paliar estas deficiencias se ha propuesto un modelo de ciclo de vida basado en la construcción de *prototipos*.

### Candidatos

En primer lugar, hay que ver si el sistema que tenemos que desarrollar es un buen candidato a utilizar el paradigma de ciclo de vida de construcción de prototipos.

- ↳ En general, cualquier aplicación que presente mucha interacción con el usuario, o que necesite algoritmos que puedan construirse de manera evolutiva, yendo de lo mas general a lo más específico es una buena candidata.

No obstante, *hay que tener en cuenta la complejidad*: 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. Puede ser que el cliente ‘no tenga tiempo para andar jugando’ o ‘no vea las ventajas de este método de desarrollo’.

- ↳ También 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 bastante frecuente que el producto de ingeniería desarrollado presente un buen rendimiento durante la fase de pruebas realizada por los ingenieros antes de entregarlo al cliente (pruebas que se realizarán normalmente con unos pocos registros en la base de datos o un único terminal conectado al sistema), pero que sea muy ineficiente, o incluso inviable, a la hora de almacenar o procesar el

volumen real de información que debe manejar el cliente. 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 la E/S 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 una cáscara vacía.

Según esto un prototipo puede tener alguna de las tres formas siguientes:

- un prototipo, en papel o ejecutable en ordenador, que describa la interacción hombre-máquina y los listados del sistema.
- un prototipo que implemente algún(os) subconjunto(s) de la función requerida, y que sirva para evaluar el rendimiento de un algoritmo o las necesidades de capacidad de almacenamiento y velocidad de cálculo del sistema final.
- un programa que realice en todo o en parte la función deseada pero que tenga características (rendimiento, consideración de casos particulares, etc.) que deban ser mejoradas durante el desarrollo del proyecto.

La secuencia de tareas del paradigma de construcción de prototipos puede verse en la figura 4.

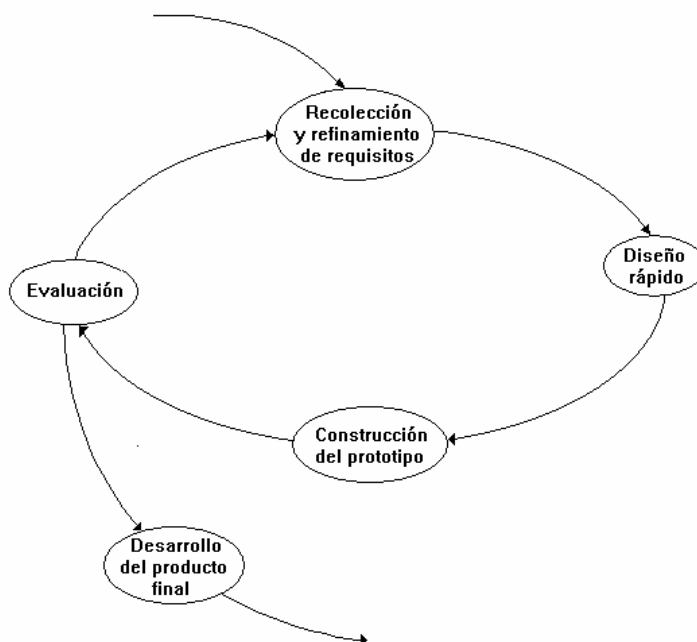


Fig. 4. Construcción de prototipos

### Fases

1. Si se ha decidido construir un prototipo, lo primero que hay que hacer es 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.
2. Luego se procede a diseñar el prototipo. 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 procedimentales de los programas: nos fijaremos más en la forma y en la apariencia que en el contenido.
3. A partir del diseño construiremos el prototipo. Existen herramientas especializadas en generar prototipos ejecutables a partir del diseño. Otra opción sería utilizar técnicas de cuarta generación. En cualquier caso, el objetivo es siempre que la codificación sea rápida, aunque sea en detrimento de la calidad del software generado.
4. Una vez listo el prototipo, hay que presentarlo 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.
5. A partir de estos comentarios del 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 entonces empezar con el desarrollo del producto final.

Por tanto, el prototipado es una técnica que sirve fundamentalmente para la fase de análisis de requisitos, pero lleva consigo la obtención de una serie de subproductos que son útiles a lo largo del desarrollo del proyecto:

- Gran parte del trabajo realizado durante la fase de diseño rápido (especialmente la definición de pantallas e informes) puede ser utilizada durante el diseño del producto final.
- Durante la fase de construcción de prototipos será necesario codificar algunos componentes del software que también podrán ser reutilizados en la codificación del producto final, aunque deban de ser optimizados en cuanto a corrección o velocidad de procesamiento.

No obstante, hay que tener en cuenta que el prototipo no es el sistema final, puesto que normalmente apenas es utilizable. Será demasiado lento, demasiado grande, inadecuado para el volumen de datos necesario, contendrá errores (debido al diseño rápido), será demasiado general (sin considerar casos particulares, que debe tener en cuenta el sistema final) o estará codificado en un lenguaje o para una máquina inadecuadas, o a partir de componentes software previamente existentes. No hay que preocuparse de haber desperdiciado tiempo o esfuerzos construyendo prototipos que luego habrán de ser desechados, si con ello hemos conseguido tener más clara la especificación del proyecto, puesto que el tiempo perdido lo ahorraremos en las fases siguientes, que podrán realizarse con menos esfuerzo y en las que se cometerán menos errores que nos obliguen a volver atrás en el ciclo de vida.

Hay que tener en cuenta que un análisis de requisitos incorrecto o incompleto, cuyos errores y deficiencias se detecten a la hora de las pruebas o tras entregar el software al cliente, nos obligará a repetir de nuevo las fases de análisis, diseño y codificación, que habíamos realizado cuidadosamente, pensando que estábamos desarrollando el producto final. Al tener que repetir estas fases, sí que estaremos desechando una gran cantidad de trabajo, normalmente muy superior al esfuerzo de construir un prototipo basándose en un diseño rápido, en la reutilización de trozos de software preexistentes y en herramientas de generación de código para informes y manejo de ventanas.

### **Problemas:**

- Con demasiada frecuencia el prototipo pasa a ser parte del sistema final, bien sea por presiones del cliente, que quiere tener el sistema funcionando lo antes posible o bien porque los técnicos se han acostumbrado a la máquina, el sistema operativo o el lenguaje con el que se desarrolló el prototipo. Se olvida aquí que el prototipo ha sido construido de forma acelerada, sin tener en cuenta consideraciones de eficiencia, calidad del software o facilidad de mantenimiento, o que las elecciones de lenguaje, sistema operativo o máquina para desarrollarlo se han hecho basándose en criterios como el mejor conocimiento de esas herramientas por parte los técnicos que en que sean adecuadas para el producto final.

El utilizar el prototipo en el producto final conduce a que éste contenga numerosos errores latentes, sea ineficiente, poco fiable, incompleto o difícil de mantener. En definitiva a que tenga poca calidad, y eso es precisamente lo que queremos evitar aplicando la ingeniería del software.

- Otro problema sería que al sustituir el prototipo por el sistema final y desechar aquel perdiéramos toda la documentación existente sobre los requisitos del sistema si únicamente estaban recogidos en el prototipo y no existía otro tipo de documentación.

## **2.7.- EL 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 complejos, mucho más realista que el ciclo de vida clásico, y permite la utilización de prototipos en cualquier etapa de la evolución del proyecto.

Otra característica de este modelo es que incorpora en el ciclo de vida el **análisis de riesgos**. Los prototipos se utilizan como mecanismo de reducción del riesgo, permitiendo finalizar el proyecto antes de haberse embarcado en el desarrollo del producto final, si el riesgo es demasiado grande.

El modelo en espiral define cuatro tipos de actividades, y representa cada uno de ellos en un cuadrante:

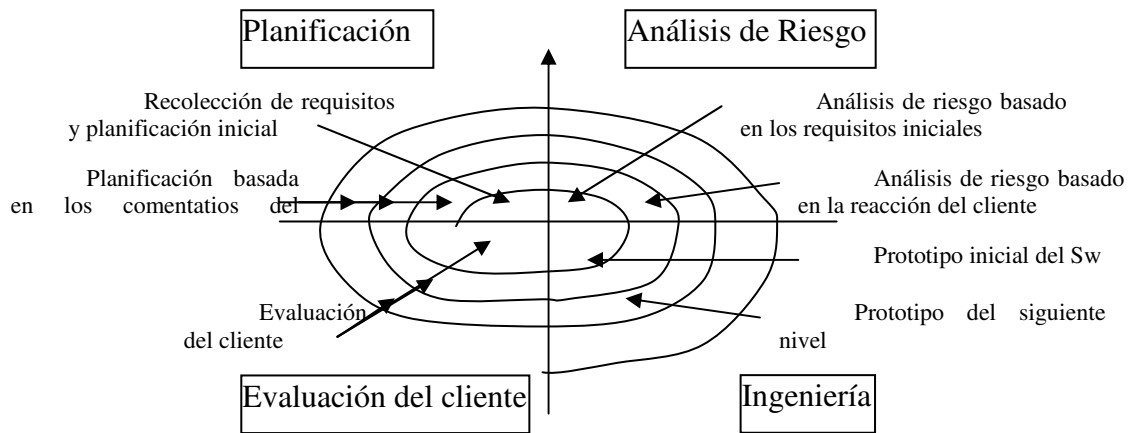


Fig. 5. El modelo en espiral.

### 2.7.1.- 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.

### 2.7.2.- ANÁLISIS DE RIESGO.

El desarrollo de cualquier proyecto complejo lleva implícito una serie de riesgos.

El análisis de riesgos consiste en cuatro actividades principales:

**Identificar los riesgos.** Pueden ser: riesgos del proyecto (presupuestarios, de organización, del cliente. etc.), riesgos técnicos (problemas de diseño, codificación, mantenimiento), riesgos del negocio (riesgos de mercado: que se adelante la competencia o que el producto no se venda bien).

**Estimación de riesgos.** Consiste en evaluar, para cada riesgo identificado, la probabilidad de que ocurra y las consecuencias, es decir, el coste que tendrá en caso de que ocurra.

**Evaluación de riesgos.** Consiste en establecer unos niveles de referencia para el incremento de coste, 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.

**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 (aquéllas que más riesgo encierran).

### 2.7.3.- INGENIERÍA.

Consiste en el desarrollo del sistema o de un prototipo del mismo.

#### 2.7.4.- 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 del sistema. Entonces el cliente procede a evaluar el prototipo y con sus comentarios, se procede a refinar los requisitos y a reajustar la planificación inicial, volviendo a empezar el ciclo.

En cada iteración 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 seguir avanzando.

Con cada iteración, se construyen sucesivas versiones del software, cada vez más completas, y aumenta la duración de las operaciones del cuadrante de ingeniería, obteniéndose al final el sistema de ingeniería completo.

La diferencia principal con el modelo de construcción de prototipos, es que en éste los prototipos se usan para perfilar y definir los requisitos. Al final, el prototipo se desecha y comienza el desarrollo del software siguiendo el ciclo clásico. En el modelo en espiral, en cambio, los prototipos son sucesivas versiones del producto, cada vez más detalladas (el último es el producto en sí) y constituyen el esqueleto del producto de ingeniería. Por tanto deben construirse siguiendo estándares de calidad.

### 2.8.- INGENIERÍA INVERSA Y REINGENIERÍA DEL SOFTWARE.

Los modelos descritos en las secciones anteriores dan una visión *directa* de la IS como un proceso evolutivo en el que el software pasa por etapas sucesivas de gestación, construcción, uso y sustitución. Sin embargo existen determinadas tareas dentro de la IS que obligan a realizar el camino inverso, lo que ha llevado a acuñar el término de ingeniería inversa.

Como ya habíamos visto en el tema anterior, en sus primeros tiempos, e incluso recientemente, el software se desarrollaba sin ninguna planificación, lo que ha llevado a la existencia de numerosas aplicaciones actualmente en funcionamiento que carecen por completo de documentación. Con el tiempo, y con la marcha de las personas que las han programado, desaparece por completo el conocimiento que se tiene sobre estas aplicaciones, y después de años de procesamiento automático de la información, incluso las personas que las manejan desconocen cuáles son exactamente las transformaciones que realizan sobre los datos que manejan.

Esto causa graves problemas a la hora de intentar realizar cualquier cambio en estas aplicaciones, al intentar optimizarlas o adaptarlas a para que funcionen sobre nuevos ordenadores, sistemas operativos o lenguajes o a la hora de reescribirlas según los nuevos estilos de trabajo con ordenador (primero fue la entrada de datos interactiva usando pantallas de texto, hoy son los entornos gráficos de ventanas).

La **ingeniería inversa** consiste en analizar un programa en un esfuerzo de representarlo en un mayor nivel de abstracción que el código fuente, de forma que se extraiga información del diseño de datos, de la arquitectura y del detalle procedimental del mismo, para poder entenderlo.

La **reingeniería del software** no sólo recupera información sobre el diseño de un programa existente sino que utiliza esta información para reestructurar o reconstruir el programa existente, con vistas a adaptarlo a un cambio, a ampliarlo o a mejorar su calidad general, con el objetivo de conseguir una mayor facilidad de mantenimiento en el futuro (esto es lo que se denomina mantenimiento preventivo).

SEWANSI

## 3.1.- INTRODUCCIÓN

---

Hay que aclarar una confusión entre los **términos metodología, método y ciclo de vida**, ya que hay autores (como Yourdon) que los utilizan indistintamente. Una metodología puede seguir uno o varios ciclos de vida, esto es, el ciclo de vida indica qué es lo que hay que obtener a lo largo del desarrollo del proyecto, pero no cómo. Esto sí lo debería indicar la metodología.

Una metodología es un concepto más amplio que método. Así, podemos considerarla como un conjunto de métodos.

### 3.1.1.- CONCEPTOS GENERALES

**Metodología de desarrollo:** se puede considerar como un conjunto de procedimientos, técnicas, herramientas, y un soporte documental que ayuda a desarrolladores a realizar nuevo software. Representa el camino para desarrollar software de una manera sistemática.

Normalmente consistirá en un conjunto de fases descompuestas en subfases (módulos, etapas, pasos, etc.). Esta descomposición del proceso de desarrollo guía a los desarrolladores en la elección de las técnicas que debe elegir para cada estado del proyecto, y facilita la planificación, gestión, control y evaluación de los proyectos.

De forma general, podemos identificar tres necesidades principales que se intentan cubrir con una metodología:

- Mejores aplicaciones. El seguimiento de una metodología es necesario pero no suficiente para asegurar la calidad del producto final.
- Un mejor proceso de desarrollo que identifica las salidas (o productos intermedios) de cada fase de forma que se pueda planificar y controlar el proyecto.
- Un proceso estándar en la organización, lo que aporta claros beneficios (por ejemplo, una mayor integración entre los sistemas y una mayor facilidad en el cambio del personal de un proyecto a otro)

Todos los entornos de desarrollo de software, aunque suelen tener objetivos similares, tienen formas de trabajo muy diferentes. Por ello la organización de desarrollo tiene dos opciones:

- Seleccionar entre un gran número de posibilidades y combinaciones de métodos de gestión, técnicas de desarrollo y soporte automatizado, para crear y desarrollar la metodología sw más apropiada.
- Analizar y evaluar las metodologías existentes y adoptar en la organización la que más se ajuste a sus necesidades. En general, esta opción es la más común.

La metodología también está influenciada por consideraciones como el tamaño y estructura de la organización, así como el tipo de aplicaciones que va a desarrollar. Por ello, no es razonable pensar que dos organizaciones utilicen la misma metodología sin realizar cambios sobre ella, ajustándola a su organización y sus proyectos. Ésta ha sido la causa de la existencia de muchas metodologías.

### 3.2.- CARACTERÍSTICAS PRINCIPALES DE LAS METODOLOGÍAS

---

Una metodología de desarrollo debería incluir una serie de características deseables entre las que destacamos las siguientes:

- **Existencia de reglas predefinidas.**

La metodología debería indicar formalmente unas reglas que definan sus fases, las tareas, los productos intermedios, las técnicas y herramientas, las ayudas al desarrollo y los formatos de documentos estándares.

- **Cobertura total del ciclo de desarrollo.**

Debe indicar los pasos que hay que realizar desde el planteamiento de un sistema hasta su mantenimiento, proporcionando mecanismos para integrar los resultados de una fase a la siguiente, de forma que pueda referenciar a fases previas y comprobar el trabajo realizado.

- **Verificaciones intermedias.**

Debe contemplar la realización de verificaciones sobre los productos generados en cada fase para comprobar su corrección. Se realizan por medio de revisiones de sw, que detectan las inconsistencias, inexactitudes, o cualquier otro tipo de defecto que se genera durante el proceso de desarrollo, evitando que salgan a relucir en la fase de pruebas o de aceptación o, aún peor, durante la fase de mantenimiento, donde las correcciones de los defectos son mucho más costosas que en las fases iniciales.

- **Planificación y control.**

Debe proporcionar una forma de desarrollar sw de manera planificada y controlada, para que no se disparen los costes ni se amplíen los tiempos de entrega. También debería ser posible incorporar una técnica de control de proyectos, aunque se piensa que este aspecto no debe necesariamente formar parte de la metodología.

- **Comunicación efectiva.**

Debería proporcionar un medio de comunicación efectiva entre los desarrolladores para facilitar el trabajo en grupo y con los usuarios.

- **Utilización sobre un abanico amplio de proyectos.**

Debe ser flexible, para que pueda emplearse sobre un amplio abanico de proyectos, tanto en variedad, tamaño y entorno. Una organización no debería usar metodologías diferentes para cada proyecto, sino que la metodología se debe poder amoldar a un proyecto concreto.

- **Fácil formación.**

La metodología la utiliza todo el personal de desarrollo de una organización, por lo que los desarrolladores deben comprender las técnicas y los procedimientos de gestión. La organización debe formar a su personal en todos los procedimientos definidos por la metodología.



- **Herramientas CASE.**

Debe estar soportada por herramientas automatizadas que mejoren la productividad del equipo de desarrollo y la calidad de sus productos resultantes. Como una metodología define las técnicas que hay que seguir en cada tarea, es necesario disponer de una herramienta que soporte, en mayor o menor grado, la automatización de dichas tareas. Esto supedita la elección de la herramienta, principalmente, a la metodología existente.

- **La metodología debe contener actividades que mejoren el proceso de desarrollo**

Uno de los principios claros de la metodología del sw es el compromiso claro sobre la mejora del proceso de desarrollo. Para ello, es necesario disponer de datos que muestren la efectividad de la aplicación del proceso sobre un determinado producto. Para ello, es necesario realizar mediciones que indiquen la calidad y el coste asociado a cada etapa del proceso, idealmente soportadas por herramientas CASE. Estos datos se deben utilizar para analizar y modificar el proceso para su mejora.

- **Soporte al mantenimiento.**

Las metodologías tradicionales se enfocaban en el desarrollo de los sistemas. Pero una vez llegado el mantenimiento, no se consideraban técnicas para la evolución lógica del sistema, que se encuentra en un entorno cambiante debido principalmente a los cambios tecnológicos y a las nuevas necesidades de los usuarios. Actualmente, el campo de la reingeniería del sw debería ser tomado en cuenta por las metodologías para facilitar, en el mayor grado posible, las modificaciones sobre los sistemas existentes.

- **Soporte de la reutilización de software.**

Las metodologías estructuradas existentes no proporcionan mecanismos para la reutilización de componentes sw. Se deberían incluir procedimientos para la creación, mantenimiento y recuperación de componentes reutilizables que no se limiten sólo al código.

### 3.3.- CLASIFICACIÓN DE LAS METODOLOGÍAS

Para poder realizar una clasificación, hay que atender a dos dimensiones o puntos de vista, como podemos ver en la tabla siguiente:

ENFOQUE	TIPO DE SISTEMA
ESTRUCTURADAS ➤ Orientadas a Procesos ➤ Orientadas a Datos ➤ Mixtas	GESTIÓN
ORIENTADAS A OBJETOS	TIEMPO REAL

Se pueden pensar metodologías combinando las dimensiones: enfoque y tipo de sistema.

### 3.3.1.- METODOLOGÍAS ESTRUCTURADAS

Las metodologías estructuradas proponen la creación de modelos del sistema que representen los procesos, los flujos y la estructura de los datos de una manera descendente (top-down). Se pasa de una visión más general del problema (un nivel alto de abstracción más cercano a las personas) hasta llegar a un nivel de abstracción más sencillo (más cercano al hardware). Esta visión se puede enfocar en las funciones (o procesos) del sistema, en la estructura de los datos, o en ambos aspectos, dando lugar a los siguientes tipos de metodologías:

- Orientadas a procesos.
- Orientadas a datos.
- Mixtas
- **Metodologías orientadas a procesos.**

La ingeniería del swf está fundada sobre el modelo básico de *entrada/proceso/salida* de un sistema. Los datos se introducen en el sistema y éste responde ante ellos transformándolos para obtener las salidas. Este modelo básico lo utilizan todas las metodologías estructuradas. Estas metodologías se enfocan fundamentalmente en la parte del **proceso** y, por esto, se describen como un enfoque de desarrollo de software orientado al proceso.

Como ejemplos de metodologías orientadas a procesos, tenemos las de [DEMARCO, 1979], [GANE Y SARSON, 1979] y, posteriormente, [YOURDON, 1989], que se basan en la utilización de un método descendente de descomposición funcional para definir los requisitos del sistema. Se apoyan en técnicas gráficas dando lugar a un nuevo concepto que es la especificación estructurada.

Una especificación estructurada es un modelo gráfico, particionado, descendente y jerárquico de los procesos del sistema y de los datos utilizados por los procesos. Se compone de:

- *Diagramas de Flujo de Datos (DFD)*. Son diagramas que representan los procesos (funciones) que debe llevar a cabo un sistema a distintos niveles de abstracción y los datos que fluyen entre las funciones. Los procesos más complejos se descomponen en nuevos diagramas hasta llegar a procesos sencillos. Es la técnica más importante del análisis estructurado, y se emplea en todas las metodologías de análisis y diseño estructurado.
  - *Diccionario de Datos*. Es el conjunto de las definiciones de todos los datos que aparecen en el DFD, tanto almacenados como en los flujos de datos.
  - *Especificaciones de proceso*. Describen con más detalles lo que ocurre dentro de un proceso, es decir, definen cómo se obtienen las salidas del proceso a partir de sus entradas.
- **Metodologías orientadas a datos.**

Dentro del modelo básico de *entrada/proceso/salida* de un sistema, ésta se orientan más a las entradas y salidas. Primero se definen las estructuras de datos y, a partir de éstas, se derivan los componentes procedimentales.

Las metodologías basadas en la información se centran en la creencia de que los datos (tipos de datos) constituyen el corazón del sistema de información, ya que son más estables que los procesos que actúan sobre ellos. La metodología que identifique con éxito la naturaleza de los datos de una organización partirá de una base estable para construir sistemas de información. Los procesos también se estudian en este tipo de metodologías, pero vienen derivados de una definición inicial de los datos, que se

representan por medio de un modelo de los datos utilizados por la organización. Este modelo estará formado por el conjunto de entidades de datos básicas y las interrelaciones entre ellas. Con este enfoque, todos los sistemas construidos están integrados sobre un mismo modelo de datos.

Un ejemplo de este tipo puede ser la metodología Ingeniería de la Información (Information Engineering- IE), que parte de una metodología de modelado de datos publicada por [MARTIN y FINKELSTEIN,1981]

### 3.3.2.- METODOLOGÍAS ORIENTADAS A OBJETOS

Se puede observar un cambio filosófico entre las metodologías clásicas de análisis y diseño estructurado y las de orientación a objeto. En las primeras, se examinan los sistemas desde el punto de vista de las funciones o tareas que deben realizar, tareas que se van descomponiendo sucesivamente en otras tareas más pequeñas y que forma los bloques o módulos de las aplicaciones. En la orientación a objetos, por su parte, cobra mucha más importancia el aspecto de *modelado* del sistema, examinando el dominio del problema como un conjunto de objetos que interaccionan entre sí.

### 3.3.3.- SISTEMAS DE TIEMPO REAL

Existen sistemas muy dependientes del tiempo que procesan información más orientada al control que orientada a los datos. Éstos son los sistemas de tiempo real, que controlan y son controlados por medio de eventos externos. Una definición de sistema de tiempo real puede ser: aquel que controla un ambiente recibiendo datos, procesándolos y devolviéndolos con la suficiente rapidez como para influir en dicho ambiente en ese momento.

Se caracterizan porque:

- Se lleva a cabo el proceso de muchas actividades de forma simultánea
- Se asignan prioridades a determinados procesos.
- Se interrumpe una tarea antes de que concluya, para comenzar otra de mayor prioridad
- Existe comunicación entre tareas
- Existe acceso simultáneo a datos comunes, tanto en memoria como en almacenamiento secundario.

No hay que confundir los sistemas de tiempo real y los sistemas en línea (interactivos). Estos últimos suelen interactuar con las personas, mientras que los de tiempo real normalmente interactúan tanto con personas como con dispositivos físicos del mundo exterior. El principal problema de estos sistemas es que si el sistema no responde con la suficiente rapidez, el ambiente puede quedar fuera de control, perdiendo datos de entrada. En un sistema en línea no se pierden datos, sólo hay que esperar un poco más.

Ejemplos típicos de tiempo real: sistemas de navegación aérea, redes de comunicaciones, control de procesos de fabricación, etc. También puede haber subsistemas de tiempo real empotrados en aplicaciones de gestión como algunos de los sistemas de gestión bancarios.

Los trabajos más importantes en este campo provienen de [WARD y MELLOR, 1985] y de [HATLEY y PIRBHAY, 1987], que permiten al analista representar el flujo y el procesamiento de control así como el flujo y procesamiento de los datos.

---



### 3.4.- PRINCIPALES METODOLOGÍAS DE DESARROLLO

---

Las administraciones públicas de distintos países europeos han promovido el desarrollo de metodologías para unificar la forma de desarrollar sus sistemas de información. Las más conocidas son MERISE, SSADM y **MÉTRICA**.

---

#### **BIBLIOGRAFÍA:**

-  *Aplicaciones Informáticas de Gestión*  
Mario G. Piattini  
RA-MA
-  *Análisis Estructurado Moderno*  
Yourdon, E.  
Prentice may