



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

Índice Tema 1

Introducción.

1. La evolución del software y la crisis del software.
2. La ingeniería del software.
 - 2.1. Principios de la ingeniería del software.
3. Fases del proceso de desarrollo del software.
 - 3.1. Fase de definición.
 - 3.2. Fase de desarrollo.
 - 3.3. Fase de mantenimiento.
4. Concepto de ciclo de vida.
5. Modelos de ciclo de vida.
6. El modelo de ciclo de vida clásico o en cascada.
 - 6.1. Las fases del modelo en cascada.
 - 6.1.1. Planificación del sistema.
 - 6.1.2. Especificación de requisitos.
 - 6.1.3. Diseño.
 - 6.1.4. Calificación.
 - 6.1.5. Pruebas e integración.
 - 6.1.6. Implantación y aceptación del sistema.
 - 6.1.7. Mantenimiento del sistema.
7. Los modelos de prototipado.
 - 7.1. El prototipado clásico.
 - 7.2. El prototipado evolutivo.

- 7.2.1. Un caso de prototipado evolutivo: el ciclo de vida RAD.
- 7.3. El modelo de desarrollo incremental.
- 8. El modelo de ciclo de vida en espiral.
 - 8.1. Operativa del modelo en espiral.
 - 8.2. Ventajas e inconvenientes del modelo en espiral.
- 9. Modelos basados en transformaciones.
 - 9.1. El modelo de técnicas de cuarta generación.
 - 9.2. El modelo de transformación de McClure.
- 10. Ciclo de vida con reutilización.
- 11. Gestión del ciclo de vida del software.



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

TEMA 1

Concepto del ciclo de vida de los sistemas y fases. Modelo en cascada y modelo en espiral del ciclo de vida.

INTRODUCCIÓN.

Un sistema informático, es decir, el conjunto de elementos que hacen posible el tratamiento automático de la información, está constituido por:

- El componente físico o hardware, que proporciona la capacidad de proceso del sistema y la interface con el mundo exterior.
- El componente lógico o software, que se puede definir como el conjunto integrado por las instrucciones o programas que cuando se ejecutan suministran la función y el comportamiento deseado, las estructuras de datos, que facilitan a los programas la manipulación adecuada de la información, y los documentos, que describen la operación y el uso de los programas.
- Y el componente humano, formado por las personas que participan en la dirección, diseño, desarrollo, implantación y explotación del sistema informático.

El objeto de este tema es el estudio del ciclo de vida de un sistema informático y, más en concreto, de su componente software.

La idea intuitiva del ciclo de vida de cualquier cosa es el conjunto de etapas o de fases por las que pasa dicha cosa desde que nace hasta que muere. En nuestro caso la cosa es el software y por tanto el ciclo de vida será el conjunto de fases por las que pasa el software desde que se concibe (se detecta una necesidad) hasta que se retira del servicio. Quiere esto decir que el concepto de ciclo de vida es inseparable del proceso de desarrollo del software y, por tanto, antes de llegar al estudio como tal del ciclo de vida del software y de los diferentes modelos que se han propuesto para representarlo, es preciso conocer primero, siquiera someramente, cómo ha evolucionado a lo largo del tiempo el desarrollo del software, la problemática que surgió paralela a esa evolución (crisis del software) y la solución a dicha problemática mediante el nacimiento de una nueva disciplina conocida como «ingeniería del software».

A partir de este momento, ya estaremos en condiciones de presentar las fases de que consta el proceso de desarrollo del software y de abordar el estudio del ciclo de vida del software. A estos efec-

tos, una vez definido el concepto de ciclo de vida, nos centraremos en el estudio de los modelos principales, tanto tradicionales, o si se quiere de más amplia utilización, como alternativos.

Dentro de los modelos tradicionales se estudiará en primer lugar el modelo de ciclo de vida clásico o modelo en cascada, sin duda el más popular, y los modelos basados en la utilización de prototipos, tanto el más clásico de prototipo desechable como los más avanzados de prototipado evolutivo y desarrollo incremental.

Entre los modelos alternativos centraremos nuestra atención en el modelo en espiral, los modelos basados en transformaciones y en los nuevos modelos de desarrollo basado en componentes, el llamado ciclo de vida con reutilización.

Finalmente el tema concluirá con una referencia a la gestión del ciclo de vida del software.

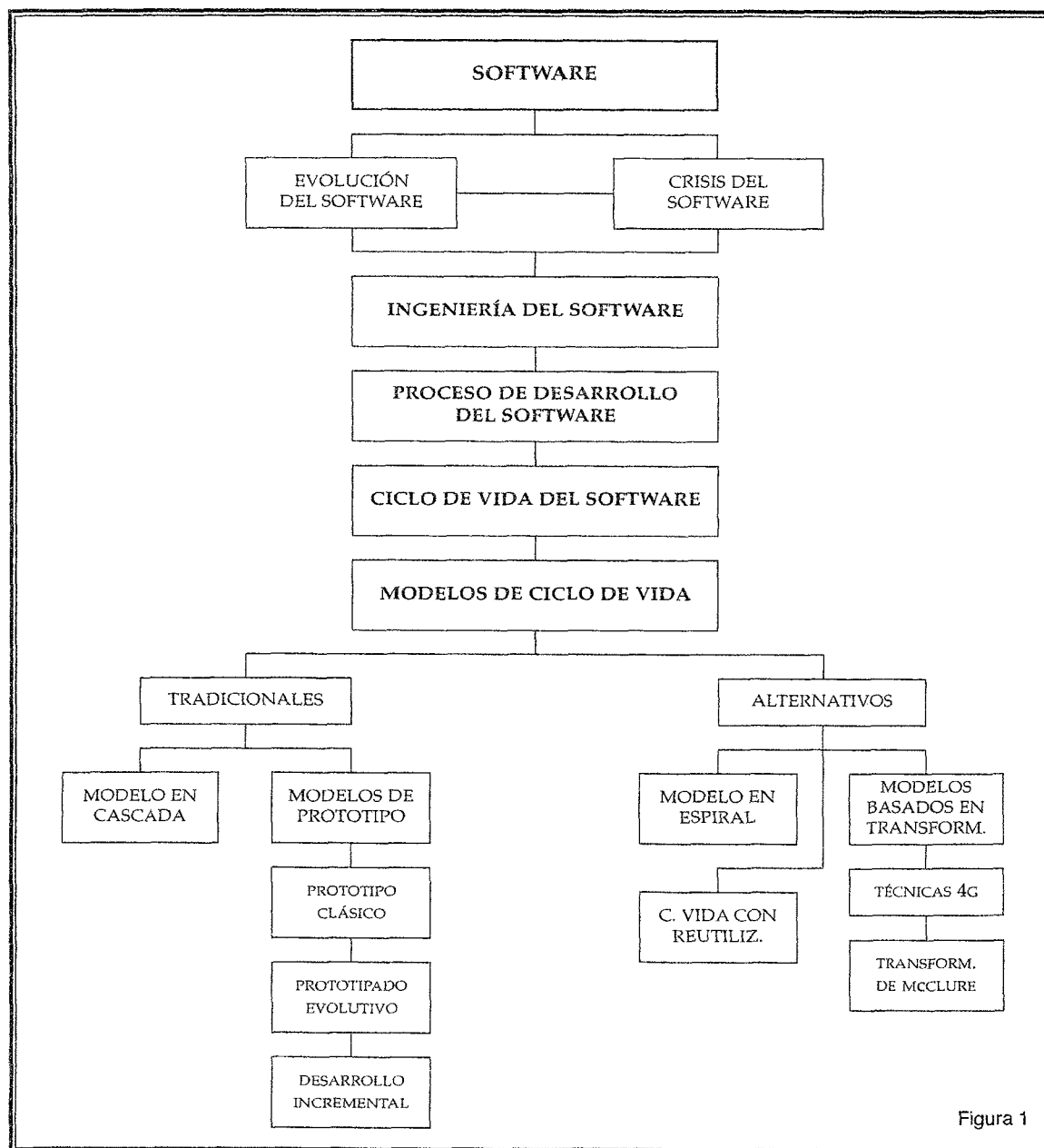


Figura 1

1. LA EVOLUCIÓN DEL SOFTWARE Y LA CRISIS DEL SOFTWARE.

Durante los primeros treinta años de la informática, el desafío principal fue desarrollar el hardware de los ordenadores de forma que se redujera el coste del procesamiento y almacenamiento de los datos. Sin embargo, a partir de la revolución microelectrónica de los años ochenta la capacidad de proceso de las máquinas ha aumentado considerablemente y han disminuido casi en la misma proporción los costes, lo que ha hecho que el reto principal de la informática se haya trasladado a reducir el coste y mejorar la calidad de las soluciones que se implementan con el software.

El contexto en que se ha desarrollado el software está muy ligado a la evolución histórica de los sistemas informáticos. Siguiendo a Pressman, se puede contemplar la evolución del software en las siguientes cuatro épocas:

- 1.^a época (1950 a 1960-65). El software se orienta al trabajo por lotes, apenas hay una distribución comercial del mismo y, prácticamente, todo el software se desarrolla a medida y generalmente sin ninguna planificación ni metodología sistemática.
- 2.^a época (1960-65 a 1975). Es la época del software orientado a los trabajos interactivos y en tiempo real y en la que aparece la primera generación de los sistemas de gestión de base de datos (SGBD). El software comienza a producirse comercialmente por distintas empresas y empieza a tomar importancia su mantenimiento, con el consiguiente gasto de recursos. Es el comienzo de la «crisis del software».
- 3.^a época (1975 a 1985-90). El software se orienta a los sistemas distribuidos, las redes de área local y las comunicaciones comienzan a adquirir gran importancia y se demanda un software que sea capaz de acceder rápidamente a los datos. Es la época del despegue de la informática; desciende el coste del hardware, aparecen los primeros ordenadores personales con el correspondiente desarrollo de paquetes para los mismos y en definitiva, el software adquiere un gran impacto en el consumidor.
- 4.^a época (a partir de 1990): es la época del software para sistemas expertos y del software de inteligencia artificial. Por otra parte, las tecnologías orientadas a objetos, las técnicas de cuarta generación (T4G) y la tecnología CASE comienzan a desplazar a otras técnicas de desarrollo del software en muchas áreas de aplicación. Se agudiza la «crisis del software» debido a la complejidad del mismo y a la creciente demanda y paralelo a ello se potencia la ingeniería del software como disciplina tendente a minimizar la crisis.

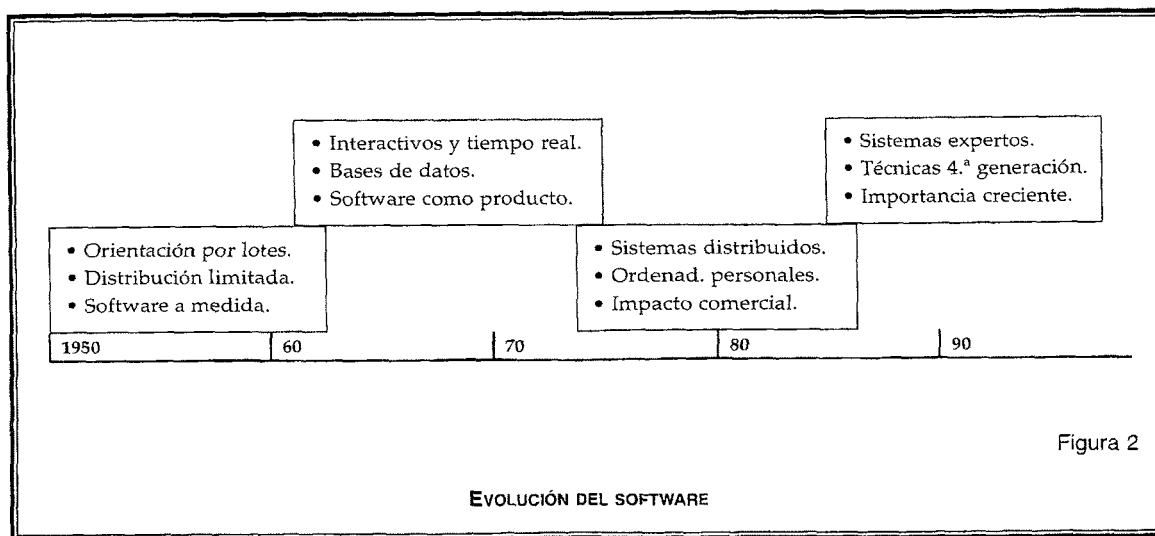


Figura 2

La rápida expansión de la Informática llevó a la escritura de millones de líneas de código antes de que se empezaran a plantear de manera seria metodologías para el diseño y la construcción de sistemas software y métodos para resolver los problemas de mantenimiento, fiabilidad, etc. Esta expansión sin control tuvo como consecuencia la denominada «crisis del software».

La «crisis del software» es el nombre genérico que se ha acuñado para referirse a un conjunto de problemas que se han ido encontrando en el desarrollo del software. Esta problemática no sólo se limita al software que no funciona adecuadamente, sino que abarca otros aspectos como:

- La forma de desarrollar el software.
- El mantenimiento de un volumen creciente de software existente.
- La forma de satisfacer la demanda creciente de software.

Los síntomas que hacen palpable la aparición de la «crisis del software» son, entre otros, los siguientes:

- Expectativas: los sistemas no responden a las expectativas que de ellos tienen los usuarios.
- Fiabilidad: los programas fallan demasiado a menudo.
- Costo: los costos del software son muy difíciles de prever y, frecuentemente, son muy superiores a lo esperado.
- Plazos: el software se suele entregar tarde y con menos prestaciones de las ofertadas.
- Portabilidad: es difícil cambiar un programa de su entorno hardware, aun cuando las tareas a realizar son las mismas.
- Mantenimiento: la modificación del software es una tarea costosa, compleja y propensa a errores.
- Eficiencia: los esfuerzos que se hacen para el desarrollo del software no hacen un aprovechamiento óptimo de los recursos disponibles (personas, tiempo, dinero, herramientas, etc.).

La solución a la «crisis del software» se centra, pues, en abordar y resolver los siguientes problemas principales:

- La planificación del proyecto software y la estimación de los costes de desarrollo, que son muy imprecisos.
- La productividad de las personas, que no se corresponde con la demanda de sus servicios.
- La calidad del producto software, que es, en muchos casos, inadecuada.

No hay una receta única que solucione la «crisis del software», sin embargo, se pueden combinar una serie de métodos para todas las fases del desarrollo del software que disciplinen el mismo. Surge así la Ingeniería del Software.

2. LA INGENIERÍA DEL SOFTWARE.

El primer reconocimiento público de la existencia de la «crisis del software» tuvo lugar en 1968 en el transcurso de una conferencia organizada por la Comisión de Ciencias de la OTAN y celebrada en Garnisch (Alemania). El objetivo de la conferencia era trazar el rumbo que permitiera salir de la crisis; en otras palabras, hacer de la construcción del software una ingeniería. Nace así una nueva disciplina, la Ingeniería del Software, que, según Bauer, se puede definir como «el establecimiento y uso de principios de ingeniería orientados a obtener, de manera económica, software que sea fiable y funcione eficientemente sobre máquinas reales».

La Ingeniería del Software abarca tres elementos clave: métodos, herramientas y procedimientos, que facilitan al gestor el control del proceso de desarrollo del software y suministran a los desarrolladores las bases para construir de forma productiva software de alta calidad.

- Los métodos proporcionan la manera de construir técnicamente el software. Abarcan las siguientes tareas:
 - Planificación y estimación de proyectos.
 - Análisis de los requerimientos del sistema y del software.
 - Diseño de las estructuras de datos, de la arquitectura de programas y de los procedimientos algorítmicos.
 - Codificación, pruebas y mantenimiento.
- Las herramientas suministran el soporte automático o semiautomático para los métodos; esto es, dan soporte al desarrollo del software.

Hoy día existen herramientas para cada método anteriormente mencionado. Cuando se integran las herramientas de forma que la información creada por una pueda ser utilizada por otra, se establece un sistema para el soporte de desarrollo del software denominado CASE.

- Los procedimientos definen la secuencia en la que se aplican los métodos, los controles que ayudan a asegurar la calidad y a coordinar los cambios y las guías que facilitan a los gestores del software establecer su desarrollo. Son el nexo de unión entre los métodos y las herramientas.

2.1. PRINCIPIOS DE LA INGENIERÍA DEL SOFTWARE.

Dado que el producto resultante de la Ingeniería del Software no es físico, las leyes físicas no sirven como fundamento. En su lugar, la Ingeniería del Software ha desarrollado unos principios basados en la observación de miles de proyectos. Algunos de estos principios, los más ampliamente aceptados, se enumeran a continuación:

1. Haz de la calidad la razón de trabajar. Independientemente de cómo se defina el término calidad, ningún cliente tolerará jamás un producto de baja calidad. La calidad se debe cuantificar y se deben establecer mecanismos de motivación y recompensa para alcanzarla.

2. Es posible el software de alta calidad. Existen técnicas que aumentan la calidad. Algunas son: la incorporación del usuario al proceso de desarrollo, el desarrollar prototipos para verificar los requisitos, la simplificación del diseño, llevar a cabo inspecciones, etc.
3. Una buena gestión es más importante que una buena tecnología. La mejor tecnología no puede compensar una mala gestión; un buen gestor puede producir buenos resultados incluso con recursos escasos.
4. Las personas y el tiempo no son intercambiables. No tiene sentido medir un proyecto sólo en base al parámetro personas/mes. El que un proyecto pueda llevarse a cabo por diez personas en un año, no significa que pueda realizarse en seis meses si intervienen veinte personas.
5. Seleccionar el modelo de ciclo de vida adecuado. No existe un modelo de ciclo de vida que funcione para cualquier proyecto. Las bases para determinar el ciclo de vida más adecuado son: la cultura de la organización, la disponibilidad para correr riesgos, el dominio de la aplicación, la volatilidad de los requisitos, etc.
6. Determinar el problema antes de escribir los requisitos.
7. Evaluar las alternativas de diseño.
8. Usar formalismos distintos para las distintas fases. Se debe seleccionar el conjunto de técnicas y formalismos adecuados para cada fase de desarrollo correspondiente. La transición entre las distintas fases es muy compleja, pero el que las fases se expresen en el mismo formalismo no lo simplifica.
9. Las técnicas son anteriores a las herramientas. Antes de usar una herramienta, se debe entender la técnica correspondiente y se debe ser capaz de seguirla.

3. FASES DEL PROCESO DE DESARROLLO DEL SOFTWARE.

Algunos autores, como Robert S. Pressman, se refieren al proceso de la Ingeniería del Software identificando tres fases genéricas que se encuentran en todo proceso de desarrollo del software, sea cual sea el área de aplicación, el tamaño del proyecto, su complejidad y el paradigma ¹ elegido.

Volviendo a lo anterior, las tres fases genéricas en todo proceso de desarrollo o construcción del software son: definición, desarrollo y mantenimiento.

3.1. FASE DE DEFINICIÓN.

Esta fase se circunscribe a dar respuesta a la pregunta «¿qué hacer?», por tanto, en esta fase se han de identificar los requerimientos clave del sistema y del software.

La fase de definición comprende tres grandes etapas: el análisis global del sistema, la planificación del proyecto software y el análisis de los requerimientos del software.

¹ Paradigma es el conjunto de pasos que hay que realizar para la utilización sistemática de los métodos, las herramientas y los procedimientos. Pressman identifica tres paradigmas básicos en Ingeniería del Software: el ciclo de vida clásico, la construcción de prototipos y las técnicas de cuarta generación.

- a) El análisis global del sistema define el papel de cada elemento del sistema informático, asignando la funcionalidad que cubrirá el software.
- b) La planificación del proyecto software consiste en asignar los recursos, estimar los costes, definir las tareas y planificar el trabajo. El propósito de esta etapa es tener una indicación preliminar de la viabilidad del proyecto en relación con su coste y con las restricciones de tiempo que se hayan establecido.
- c) El análisis de los requerimientos del software consiste en detallar todo lo posible el dominio de la información y la función que ha de cumplir el software.

3.2. FASE DE DESARROLLO.

La fase de desarrollo se circunscribe a dar respuesta a la pregunta «¿cómo hacerlo?». Dependiendo del paradigma utilizado varían los métodos de llevar a cabo el desarrollo del software, pero, de cualquier manera, han de producirse siempre tres pasos concretos: el Diseño del Software, la Codificación y las Pruebas del Software.

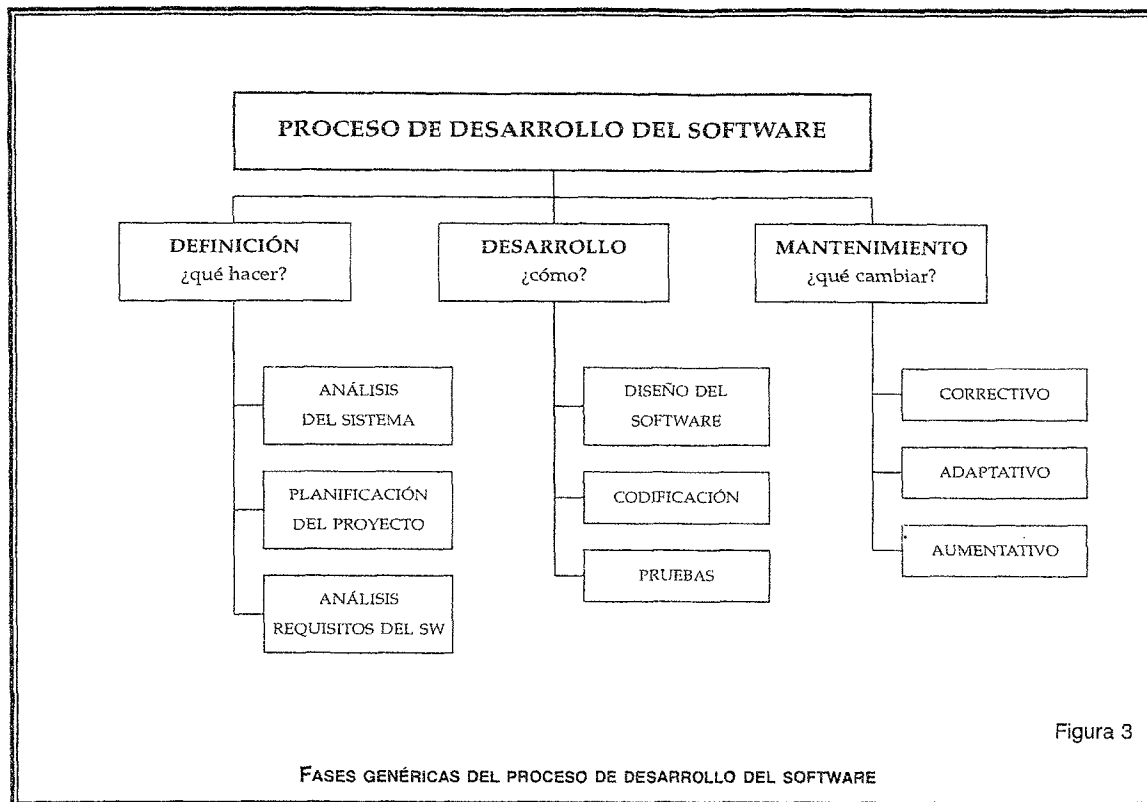
- a) El Diseño del Software consiste en trasladar los requerimientos de éste a un conjunto de representaciones que describen la estructura de los datos, la arquitectura y el procedimiento algorítmico.
- b) La Codificación consiste en trasladar las representaciones del diseño a un lenguaje que da como resultado un conjunto de instrucciones ejecutables por el ordenador.
- c) Las Pruebas del Software consisten en probar éste, una vez codificado, a fin de descubrir los defectos.

3.3. FASE DE MANTENIMIENTO.

La fase de mantenimiento se centra en la cuestión ¿qué hay que cambiar? El mantenimiento del software se enfoca sobre el cambio que va asociado a una corrección de errores, a nuevas adaptaciones requeridas por la evolución del entorno del software o a modificaciones debidas a los cambios de requerimientos del cliente. La fase de mantenimiento reapplica los pasos de las fases de definición y desarrollo del software.

Los cambios en el software pueden ser de tres tipos, dando lugar cada uno de ellos a un tipo de mantenimiento:

- a) Mantenimiento correctivo, que se realiza para corregir defectos o errores en el software.
- b) Mantenimiento adaptativo, que se produce cuando por el paso del tiempo hay cambios en el entorno original de desarrollo del software; por ejemplo, cambia el sistema operativo, etc.
- c) Mantenimiento aumentativo, que se produce cuando se amplía el software más allá de sus requerimientos funcionales originales; por ejemplo, nuevas funcionalidades requeridas por los clientes.



4. CONCEPTO DE CICLO DE VIDA.

De todo lo visto hasta aquí, puede deducirse que el proceso genérico de desarrollo del software se corresponde con una serie de actividades que comienzan con la identificación de una necesidad y concluyen con el retiro del software que satisface dicha necesidad. Este enfoque orientado al producto, o mejor, a la transformación que sufre el producto, es lo que da lugar al concepto de ciclo de vida.

Precisando más la idea de ciclo de vida, podemos definir éste como el «conjunto de etapas por las que atraviesa el sistema desde su concepción hasta su retirada de servicio, pasando por su desarrollo y explotación».

No existe un único modelo de ciclo de vida que defina los estados por los que pasa cualquier producto software. Dado que existe una gran variedad de aplicaciones y que dicha variedad supone situaciones totalmente distintas, es natural que existan diferentes modelos de ciclo de vida. No obstante, todo ciclo de vida debe cubrir los siguientes objetivos básicos:

- Definir las actividades a realizar y en qué orden, es decir, determinar el orden de las fases del proceso software.
- Establecer los criterios de transición para pasar de una fase a la siguiente.
- Proporcionar puntos de control para la gestión del proyecto, es decir, calendario y organización.
- Asegurar la consistencia con el resto de los sistemas de información de la organización.

Aunque no existe un modelo de ciclo de vida que sirva para cualquier proyecto, la utilización de un modelo de ciclo de vida para el desarrollo de los sistemas de información siempre es recomendable para cualquier tipo de proyecto y organización y especialmente en el caso de proyectos complejos, donde se verán incrementadas sus ventajas. Cada proyecto debe seleccionar el modelo de ciclo de vida que sea más apropiado para su caso, el cual se elige en base a considerar una serie de factores como: la cultura de la organización, el deseo de asumir riesgos, el área de aplicación, la volatilidad de los requisitos, la comprensión de dichos requisitos, etc.

Las funciones principales de un modelo de ciclo de vida del software son las de determinar el orden de las etapas implicadas en el desarrollo y evolución del software y las de establecer los criterios de transición para pasar desde una etapa a la siguiente. Incluye, por tanto, los criterios para determinar cuándo acaba la etapa actual y para seleccionar la etapa siguiente. Esto es, en cualquier proyecto software, el modelo de ciclo de vida permite responder a las siguientes cuestiones: ¿qué se hará a continuación? y ¿por cuánto tiempo se hará?

Dado que cada modelo de ciclo de vida tiene sus ventajas y sus inconvenientes, no se suelen seguir en la práctica los modelos en su forma pura, sino que de acuerdo con las peculiaridades del sistema y la experiencia del personal, se pueden adoptar aspectos de otros modelos que sean más adecuados al caso concreto.

Finalmente, es importante no confundir el concepto de ciclo de vida con el de metodología. Mientras que el ciclo de vida indica qué actividades hay que realizar y en qué orden, la metodología indica cómo avanzar en la construcción del sistema, esto es, con qué técnicas, y entre sus características está la de determinar los recursos a utilizar o las personas implicadas en cada actividad.

5. MODELOS DE CICLO DE VIDA.

Algunos autores de Ingeniería del Software clasifican los modelos de ciclo de vida en dos grandes grupos: modelos de ciclo de vida tradicionales y modelos de ciclo de vida alternativos.

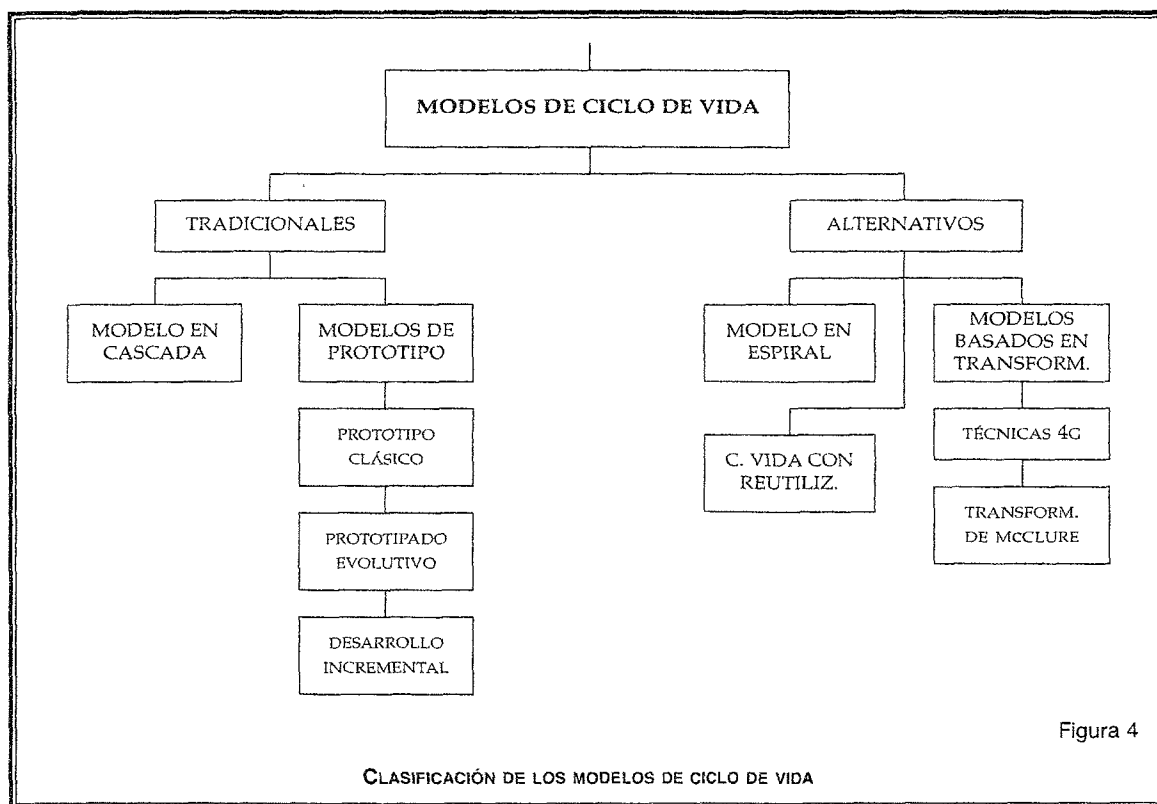
Los modelos tradicionales incluyen aquellos que existen, prácticamente, desde los primeros tiempos de la Ingeniería del Software, y están ampliamente tratados en casi todos los libros de esta disciplina. Modelos tradicionales son:

- a) El modelo de ciclo de vida clásico o en cascada, que se detallará más adelante.
- b) El modelo de ciclo de vida de refinamiento sucesivo o mejora iterativa, que es una recomendación orientada a mejorar el modelo en cascada, que predica la generación de los productos de forma iterativa, mediante un proceso de refinamiento y mejora continuos desde las especificaciones de alto nivel hasta los componentes del código fuente.
- c) Estándares militares e industriales, que son modelos de desarrollo de software (por ejemplo, la norma ESA-PSS-05-0 de la Agencia Espacial Europea para su desarrollo de software) que proponen alguna variación sobre el modelo clásico. Generalmente, ponen especial énfasis en la definición de productos entregables, revisiones, hitos y técnicas requeridas en cada caso.
- d) Los modelos basados en la utilización de prototipos, cuya utilización se ha extendido tanto y tan rápidamente que pueden considerarse como unos más de los modelos tradicionales. Más adelante se verán con detalle.

Los modelos de ciclo de vida alternativos centran su atención bien sobre productos distintos a los clásicos (utilización de componentes reutilizables, etc.), bien sobre procesos especiales de producción (automatización de la programación, gestión de riesgos, etc.), o bien sobre entornos de producción. Algunos modelos alternativos son:

- a) Los modelos orientados al desarrollo de productos software, que representan una extensión a los modelos tradicionales por la generación de productos distintos. Entre ellos cabe citar al modelo de ensamblaje de componentes reutilizables, consistente en configurar y especializar componentes de software ya existentes.
- b) Los modelos orientados al proceso de producción de software, que pueden verse bien como programas que implementan un régimen particular de inferencia y evolución del software, por ejemplo, los modelos basados en la utilización de técnicas de cuarta generación y los modelos de transformación, o bien como enfoques conceptuales, como el modelo en espiral.

Nos referiremos a continuación a algunos de estos modelos de ciclo de vida siguiendo, a efectos de ser lo más sistemáticos posible, el orden que se muestra en el siguiente esquema.



6. EL MODELO DE CICLO DE VIDA CLÁSICO O EN CASCADA.

Antes de que Royce presentara en 1970 el modelo de ciclo de vida en cascada (Waterfall Model) los modelos de desarrollo del software se basaron principalmente en el modelo CODE-AND-FIX (codifica y mejora) y en el modelo por etapas (Stage-Wise Model).

CODE-AND-FIX fue el modelo básico utilizado en los primeros tiempos de desarrollo del software. Se componía de dos pasos: Escribir algún código o programa (Code) y Resolver los problemas en el código (Fix).

Lógicamente, este modelo presentaba una serie de dificultades entre las que cabe citar: la falta de estructuración del código tras un cierto número de ajustes y su poca preparación para ser modificado, lo que hacía que resultaran muy costosos los siguientes ajustes; y, con frecuencia, la poca adaptación del software, incluso estando bien diseñado, a las necesidades del usuario.

Los problemas apuntados del modelo Code and Fix llevaron a la necesidad de realizar el desarrollo del software siguiendo un modelo de etapas sucesivas, el modelo Stage Wise que considera las siguientes: Planificación, Especificaciones de operación, Especificaciones de codificación, Codificación, Prueba de cada unidad, Prueba de integración, Eliminación de problemas, y Evaluación del sistema.

El modelo en cascada introduce una serie de mejoras respecto al modelo por etapas tales como:

- Considerar la realización de bucles de realimentación entre etapas, permitiendo que se puedan resolver los problemas detectados en una etapa, en la etapa anterior.
- Permitir la incorporación inicial del prototipado en el ciclo de vida del software, a fin de captar las especificaciones durante el análisis, o para probar distintas soluciones durante el diseño.

El modelo en cascada se compone de una serie de fases que se suceden secuencialmente, generándose en cada una de ellas unos resultados que serán necesarios para iniciar la fase siguiente. Es decir, la evolución del producto software se produce a través de una secuencia ordenada de transiciones de una fase a la siguiente, según un orden lineal.

El número de fases en este modelo es irrelevante, ya que lo que le caracteriza es la secuencialidad de las mismas y la necesidad de completar cada una de ellas para pasar a la siguiente. Por otra parte, este modelo permite iteraciones durante el desarrollo, ya sea dentro de un mismo estado, o de un estado hacia otro anterior.

El modelo en cascada tiene tres propiedades muy positivas:

1. Las etapas están organizadas de un modo lógico.
2. Cada etapa incluye cierto proceso de revisión, y se necesita una aceptación del producto antes de que la salida de la etapa pueda usarse. El modelo del ciclo de vida en cascada está regido por la documentación, es decir, la decisión del paso de una fase a la siguiente se toma en función de si la documentación asociada a dicha fase está completa o no.
3. El ciclo es iterativo, esto es, aunque el flujo básico es de arriba hacia abajo, los problemas encontrados en las etapas inferiores afectan a las decisiones de las etapas superiores.

Desde su presentación, el modelo en cascada ha tenido un papel fundamental en el desarrollo de proyectos software. Ha sido, y todavía sigue siendo, el más utilizado, tanto que este modelo se conoce con el nombre de «ciclo de vida clásico», si bien incorporando infinidad de variaciones que eliminan el carácter simplista del mismo. Aun así, existen una serie de limitaciones que justifican la necesidad de definir otros modelos.

Las principales críticas o limitaciones del modelo en cascada se centran en sus características básicas de secuencialidad y necesidad de utilizar los resultados de una fase como arranque de la siguiente, de manera que el sistema sólo se puede validar cuando está terminado.

- Respecto a la secuencialidad se aduce que existen muchos proyectos para los cuales el orden que propone el modelo en cascada es inviable.
- Respecto al segundo inconveniente se argumenta que el modelo, en su formulación pura, no prevé revisiones o validaciones intermedias por parte del usuario y los resultados sólo se ven al final de una serie de fases y tareas, de forma que si se produce un error en las primeras fases, éste sólo se detectará al final y su corrección tendrá un costo muy elevado.
- Por último, una limitación más es que el modelo en cascada asume que los requisitos de un sistema pueden ser congelados antes de comenzar el diseño, lo que, para sistemas totalmente nuevos, es poco realista y, además, requiere seleccionar previamente el hardware.

Para paliar estos inconvenientes se emplean, sobre todo, los modelos basados en la utilización de prototipos.

6.1. LAS FASES DEL MODELO EN CASCADA.

Como se ha indicado anteriormente, las fases que comprende el ciclo de vida clásico son irrelevantes, tanto en número, como en cuáles sean esas fases siempre que se produzcan secuencialmente. Tan modelo en cascada es uno de tres fases (diseño, programación e implantación y pruebas de aceptación), válido para proyectos muy pequeños y de escasa o nula complejidad, como un modelo de nueve o más etapas, adecuado a proyectos grandes.

Posiblemente, el modelo clásico más utilizado sea el modelo de siete fases que presentamos y describimos a continuación. Estas fases son:

1. Planificación.
2. Especificación de requisitos.
3. Diseño.
4. Codificación.
5. Pruebas e Integración.
6. Implantación y aceptación.
7. Mantenimiento.

6.1.1. Planificación del sistema.

Es una de las fases más importantes del ciclo de vida, puesto que de ella depende que el sistema se desarrolle de manera más efectiva en cuanto a costes y plazos de entrega. En esta fase es necesario fijar el ámbito del trabajo a realizar, los recursos necesarios, las tareas a realizar, las referencias a tener en cuenta, el coste estimado del proyecto, la composición del equipo de desarrollo y el orden de las actividades.

6.1.2. Especificación de requisitos.

En esta fase es preciso analizar, entender y documentar el problema que el usuario trata de resolver con el sistema y se han de especificar con detalle las funciones, objetivos y restricciones del mismo, a fin de que usuarios y desarrolladores puedan tomar éstas como punto de partida para acometer el resto del sistema. Es decir, en la fase de especificación de requisitos se trata de definir qué debe hacer el sistema, e identificar la información a procesar, las funciones a realizar, el rendimiento del sistema, las interfaces con otros sistemas y las ligaduras de diseño.

6.1.3. Diseño.

Arranca de las especificaciones de la fase anterior. En la fase de diseño, una vez elegida la mejor alternativa, se debe crear la solución al problema descrito atendiendo a aspectos de interfaces de usuario, estructura del sistema y decisiones sobre la implantación posterior. La fase de diseño trata de definir el «cómo» y se deberán para ello, diseñar las estructuras de datos, la arquitectura del sistema y los detalles que permitan la codificación posterior y las pruebas a realizar.

6.1.4. Codificación.

Esta fase consiste en traducir las especificaciones y representaciones del Diseño a un lenguaje de programación capaz de ser interpretado y ejecutado por el ordenador. En todo caso, el lenguaje vendrá determinado por el entorno tecnológico del sistema, y el programador deberá velar por la claridad de su estilo para facilitar cualquier interpretación posterior de los programas.

6.1.5. Pruebas e integración.

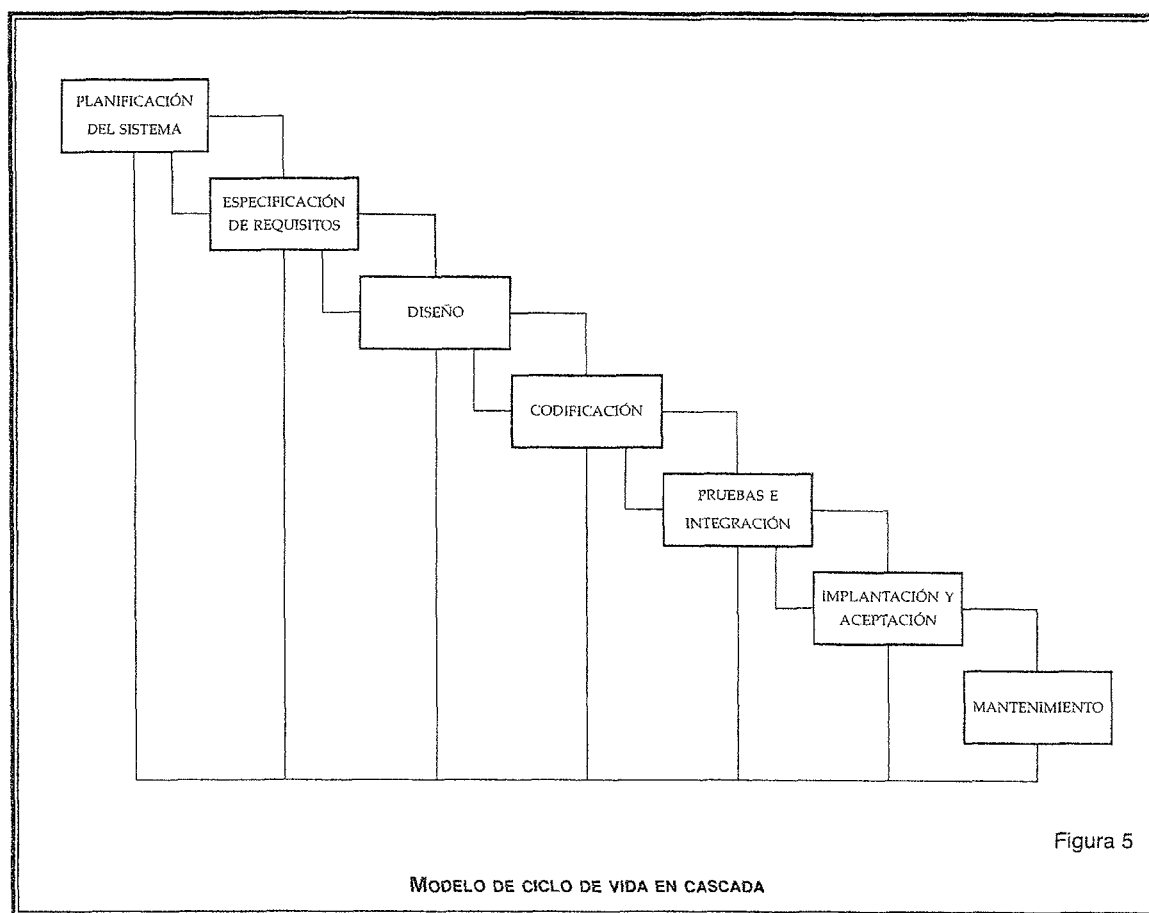
Una vez que se tienen los programas en el formato adecuado al ordenador, hay que llevar a cabo las pruebas necesarias que aseguren la corrección de la lógica interna del programa y que éste cubre las funcionalidades previstas. La integración de las distintas partes que componen la aplicación o el sistema debe garantizar el buen funcionamiento del conjunto.

6.1.6. Implantación y aceptación del sistema.

El objetivo de esta fase es conseguir la aceptación del sistema por parte de los usuarios del mismo, y llevar a cabo las actividades necesarias para su puesta en producción. Para ello, toma como punto de partida los componentes del sistema probados unitaria e integradamente en la fase anterior y se prueban de nuevo con el fin de verificar que se cumplen los requisitos de usuario y que el sistema es capaz de manipular los volúmenes de información requeridos en los tiempos y velocidades deseados, así como que funcionan las interfaces con otros sistemas.

6.1.7. Mantenimiento del sistema.

La fase de mantenimiento comienza una vez que el sistema ha sido entregado al usuario y continúa mientras permanece activa su vida útil. Como ya se ha indicado, el mantenimiento puede deberse a errores no detectados previamente (correctivo), a modificaciones, mejoras o ampliaciones solicitadas por los usuarios (perfectivo, o aumentativo) o a adaptaciones requeridas por la evolución del entorno tecnológico o cambios normativos (mantenimiento adaptativo).



7. LOS MODELOS DE PROTOTIPADO.

Los modelos basados en la utilización de prototipos fueron creados para solventar las deficiencias percibidas en el modelo en cascada, deficiencias centradas, básicamente, en la poca adaptación del modelo a los cambios.

Mientras que algunos prototipos se construyen para determinar si los requisitos son razonables o si las especificaciones son completas, otros se realizan para ver si el diseño es aún posible. Los prototipos permiten a los desarrolladores construir rápidamente tempranas versiones de los sistemas software, que pueden evaluar los usuarios. Estas evaluaciones pueden ser incorporadas como retroalimentación para refinar los diseños y especificaciones del sistema. La idea básica es que el prototipo ayude a comprender los requisitos del usuario; pero además, los prototipos pueden usarse para verificar la viabilidad del diseño del sistema, y como una herramienta iterativa del desarrollo del software donde el prototipo evoluciona hasta llegar al sistema final.

La construcción de un prototipo es un proceso que facilita al programador la creación del modelo de software a construir. Este modelo puede ser:

- Un prototipo en papel, que describa la interacción hombre-máquina de forma que facilite al usuario la comprensión de cómo se producirá el trabajo.
- Un prototipo que funcione, que implemente algunos subconjuntos de la funcionalidad requerida al software deseado, de manera que se puedan apreciar mejor las características y posibles problemas.

Existen varios modelos derivados del uso de prototipos. Algunos de ellos son:

- Prototipo clásico. Es un prototipo desechable, que se usa para ayudar al cliente a identificar los requisitos, y el que se implantan sólo aquellos aspectos del sistema que se entienden mal o son desconocidos.
- Maqueta. Aporta al usuario un ejemplo visual de entradas y salidas. Se diferencia del anterior en que mientras que aquél utiliza datos reales, las maquetas son formatos encadenados de entrada y salida con datos simples estáticos.
- Prototipo evolutivo. Aporta a los usuarios una representación física de las partes clave del sistema antes de la implantación. Una vez definidos todos los requisitos, el prototipo evolucionará hacia el sistema final. En los prototipos evolutivos se implantan aquellos requisitos y necesidades que son claramente entendidos, utilizando análisis y diseño en detalle así como datos reales.

La construcción de prototipos puede ser problemática por las siguientes razones:

1. El cliente ve funcionando lo que parece ser una versión del software, ignorando que el prototipo, en palabras de Brooks, se ha hecho con «chicle y alambres» y que por las prisas en hacer que funcione, no se han tenido en cuenta los aspectos de calidad y mantenimiento a largo plazo del software. Cuando se le informa que el producto debe ser reconstruido, el cliente no está conforme y solicita que se apliquen cuantas mejoras sean necesarias para hacer del prototipo el producto final. En muchas ocasiones se cede a esta petición.
2. El desarrollador del software adopta a menudo ciertos compromisos de implementación en orden a obtener un producto que funcione rápidamente. Puede utilizar un sistema operativo o un lenguaje de programación inapropiado simplemente porque está disponible y es conocido, o implementar algoritmos ineficientes porque lo puede hacer de forma sencilla. Después de pasar algún tiempo, el desarrollador olvida las razones por las que las elecciones que hizo eran inapropiadas, y la elección menos adecuada pasa a formar parte del sistema.

La clave está en definir al comienzo las reglas del juego; esto es, el cliente y el técnico deben estar de acuerdo en que el prototipo se construya sólo para servir como un mecanismo de definición de los requerimientos y en que posteriormente ha de desecharse y debe construirse el sistema real con los ojos puestos en la calidad y el mantenimiento.

De lo expuesto anteriormente, se pueden enumerar, a modo de resumen, las ventajas e inconvenientes que presenta el modelo de prototipado.

A) Las principales ventajas son las siguientes:

- Los requisitos de los usuarios son más fáciles de determinar, y la implantación del sistema es más sencilla debido a que los usuarios conocen lo que esperan. Además, el sistema resultante es el sistema deseado y necesita pocos cambios.
- Los sistemas se desarrollan más rápidamente, y el esfuerzo de análisis y programación y, por tanto, el coste de desarrollo, se reduce.
- El prototipado facilita la comunicación con los usuarios, en concreto, mejora la comunicación entre usuario y analista y los sistemas son más fáciles de aprender y utilizar por los usuarios finales.

B) En cuanto a los principales inconvenientes, cabe citar los siguientes:

- El prototipado crea incorrectas expectativas en el usuario, que puede ver el prototipo como sistema final.
- Se producen inconsistencias entre el prototipo y el sistema final, ya que el prototipo sólo es un paso intermedio y no tiene por qué ser idéntico al sistema final.
- Suele darse frecuentemente la intromisión de los usuarios finales en la integración.
- El prototipado no es apto para proyectos muy grandes a largo plazo, ni incluso para aplicaciones pequeñas de menos de un mes de tiempo de desarrollo. El tiempo medio de desarrollo estimado para éxitos del prototipado, según Martin, es en aplicaciones o proyectos de tamaño medio cuya duración pueda estar fijada entre tres y cinco meses.

7.1. EL PROTOTIPADO CLÁSICO.

El modelo de prototipado clásico o prototipo desechable comienza con la definición de los requerimientos del usuario (recolección de requerimientos). El técnico y el cliente se reúnen y definen los objetivos globales para el software, identifican todos los requerimientos conocidos y perfilan las áreas donde será necesaria una mayor definición. Sobre estos requerimientos se produce un «diseño rápido», que se enfoca sobre la representación de los aspectos del software visibles al usuario (p. ej., métodos de entrada y formatos de salida, etc.). El diseño rápido conduce a la construcción de un prototipo, el cual es evaluado por el cliente/usuario y se utiliza para refinar los requerimientos del software a desarrollar.

La siguiente figura ilustra este modelo.

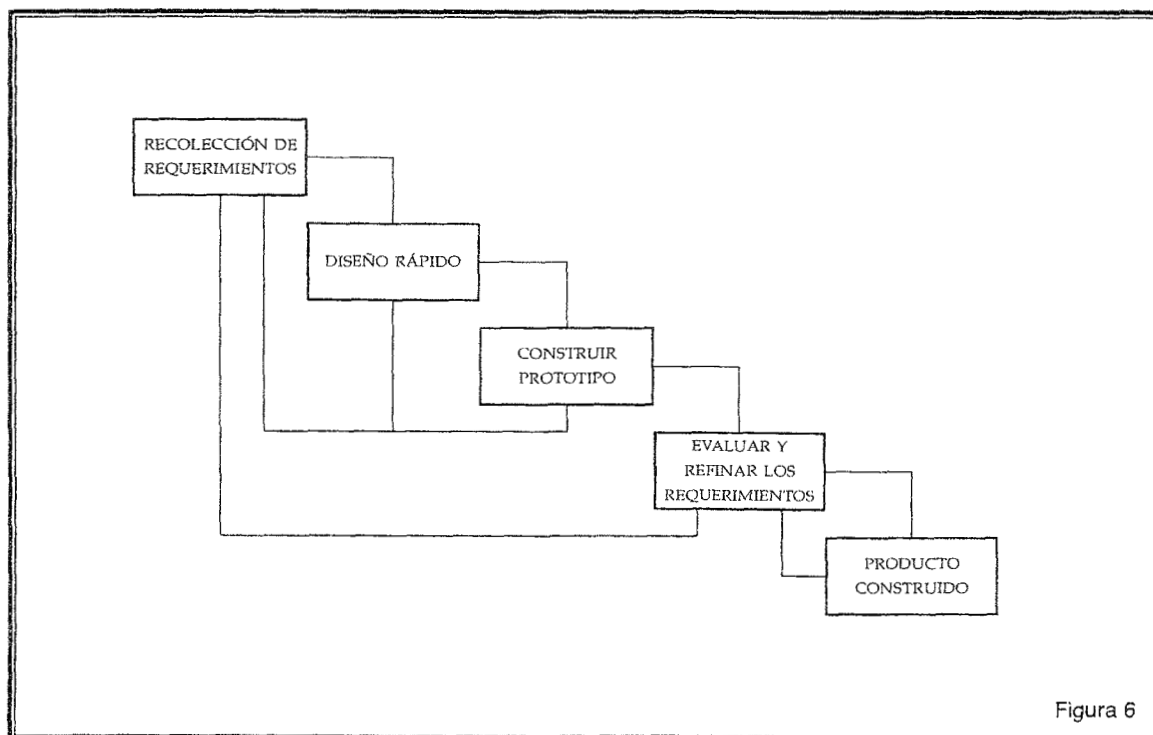


Figura 6

Este proceso es un proceso interactivo en el que el prototipo se va afinando para que satisfaga las necesidades del cliente, al mismo tiempo que facilita al que lo desarrolla una mejor comprensión de lo que hay que hacer. Si el prototipo construido es un prototipo que funciona, el desarrollador intentará hacer uso de fragmentos de programas existentes o aplicará herramientas que faciliten la rápida generación de programas que funcionen.

Es importante precisar que el prototipo se construye sólo para servir como mecanismo de definición de los requerimientos funcionales. Posteriormente ha de desecharse y debe construirse el sistema con los criterios normales de calidad y mantenimiento, siguiendo, por ejemplo, el ciclo de vida clásico.

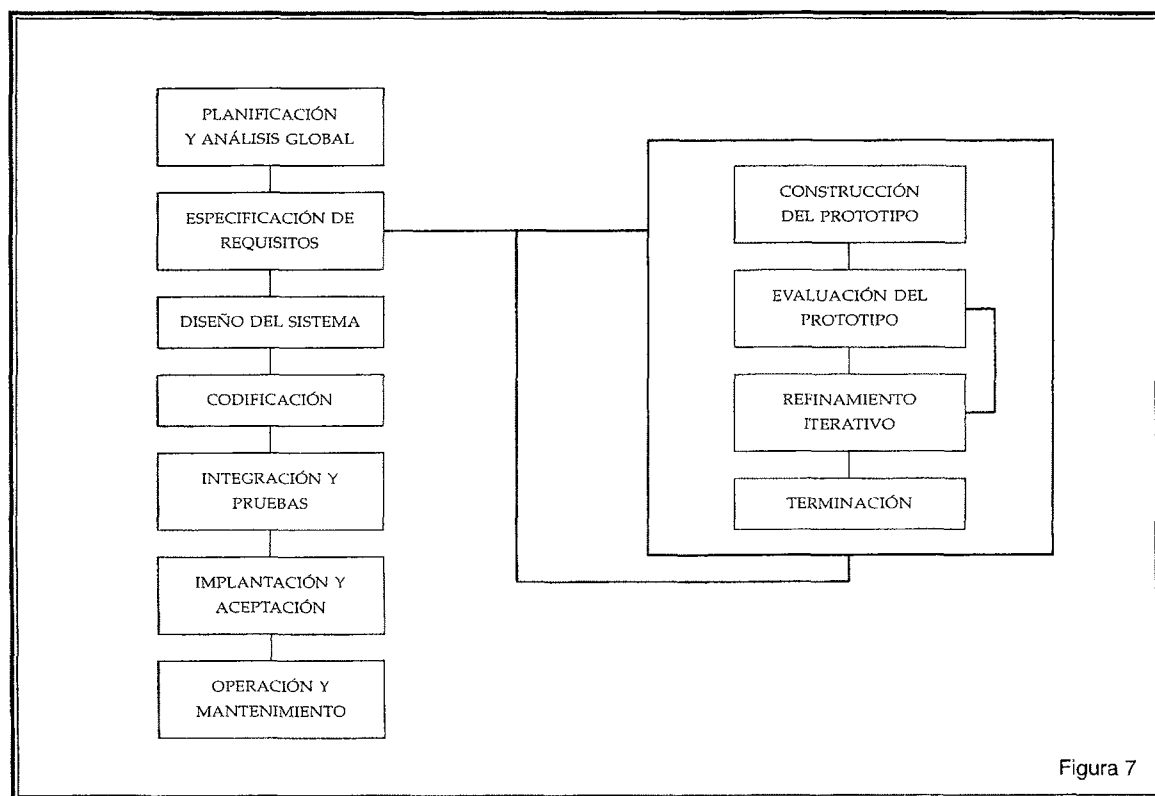


Figura 7

7.2. EL PROTOTIPADO EVOLUTIVO.

En este tipo de ciclo de vida, establecido por Jackson, se construye una implementación parcial del sistema que satisface los requisitos conocidos, la cual es utilizada por el usuario para llegar a comprender mejor la totalidad de requisitos que desea.

Desde un punto de vista genérico, se puede decir que los modelos evolutivos se encaminan a conseguir un sistema flexible que se pueda expandir, de forma que se pueda realizar rápidamente un nuevo sistema cuando cambian los requisitos. Estos modelos consisten en implementar un producto software operativo y hacerle evolucionar de acuerdo con la propia experiencia operacional. Están especialmente indicados en situaciones en que se utilizan lenguajes de cuarta generación (L4G) y para aquellas otras en que el usuario no puede decir lo que requiere, pero lo reconocerá cuando lo vea. Los modelos evolutivos dan al usuario una rápida capacidad de operación inicial y una buena base para determinar mejoras del sistema.

La diferencia fundamental entre el prototipado clásico y el evolutivo estriba en que mientras que en el primer caso se asume que existen una serie de requisitos reales, aunque para establecer lo que el usuario quiere realmente es necesario establecer una serie de iteraciones antes de que los requisitos se estabilicen al final, en el caso evolutivo se asume desde el principio que los requisitos cambian continuamente.

En el prototipo clásico o desechable lo lógico es implementar sólo aquellos aspectos del sistema que se entienden mal, mientras que en el prototipo evolutivo lo lógico es comenzar por los aspectos que mejor se comprenden y seguir construyendo apoyados en los puntos fuertes y no en los débiles. Por tanto, la idea de prototipo de construcción rápida y poco cuidadosa se corresponde con el prototipo desechable, nunca con el evolutivo.

Como resultado de este modo de desarrollo, la solución software evoluciona acercándose cada vez más a las necesidades del usuario; ahora bien, pasado un tiempo el sistema software así construido deberá ser rehecho o sufrir una profunda reestructuración con el fin de seguir evolucionando.

7.2.1. Un caso de prototipado evolutivo: el ciclo de vida RAD.

Dentro de los modelos de desarrollo evolutivo es importante destacar el modelo RAD o iterativo.

RAD son las siglas de la expresión anglosajona Rapid Application Development, esto es en español, Desarrollo Rápido de Aplicaciones.

Aunque a menudo el concepto de Desarrollo Rápido de Aplicaciones se utiliza indistintamente al de prototipado, se ajusta más, en realidad, a un subtipo del modelo de prototipado, el de Prototipo Evolutivo.

Según Martin, en su libro dedicado al RAD («La metodología formal para el prototipado») son cuatro los ingredientes que forman los elementos para el uso del prototipado: Personas, Herramientas, Metodología y Gestión.

Las personas son la clave del éxito del proyecto RAD y deben estar formadas y bien entrenadas. Por otra parte, las personas deben disponer de herramientas potentes, como lenguajes de cuarta generación o herramientas CASE, debe existir una metodología formal de trabajo y un proceso libre de burocracia para el desarrollo del prototipo.

El RAD es una metodología muy disciplinada que se estructura para su realización en pasos o fases; pero, además, se rodea de varios elementos como son: la utilización de diagramas PERT para describir la lista de acciones a realizar (gestión del proyecto); un I-CASE (CASE integrado) que permita el análisis gráfico, diseño y construcción de la aplicación; un diccionario potente; herramientas de prueba; y depuradores en la creación del prototipo.

Las fases componentes del ciclo de vida RAD son:

1. Planificación de requerimientos. Describe, a partir de los usuarios, los objetivos y el cómo de las aplicaciones. Se utiliza la forma de trabajo JRP (Joint Requirement Planning, Plani-

ficación Conjunta de Requerimientos), que se desarrolla en sesiones conjuntamente con los usuarios.

2. Diseño con el usuario. Consiste en la participación de los usuarios, de forma no técnica, en sesiones JAD (Joint Application Design, Diseño Conjunto de Aplicaciones), tanto de las interfaces como de la sensación de la aplicación.
3. Construcción. Con ayuda de un I-CASE se logra tener la aplicación generada por parte de los programadores, a veces con muy pocas modificaciones. En el caso de no disponer de herramientas CASE, se seguirán los pasos tradicionales de construcción y programación de aplicaciones, con el consiguiente retardo y aumento de esfuerzo y costes.
4. Implantación. Una vez terminada la aplicación, ésta debe pasar las pruebas y realizarse los correspondientes cursos de formación.

7.3. EL MODELO DE DESARROLLO INCREMENTAL.

Dentro de los modelos evolutivos algunos autores engloban también el modelo de desarrollo incremental, aunque ambos modelos tienen diferencias notorias.

El modelo de desarrollo incremental, que fue propuesto por Hirsch en 1985, consiste en desarrollar un sistema que satisfaga una parte de los requisitos especificados y posteriormente ir creando versiones que incorporen los requisitos que faltan, hasta llegar al sistema final.

Actuando así, se pretende disponer pronto de un sistema que aunque sea incompleto, sea utilizable y satisfaga parte de los requisitos, evitando de paso el efecto «big bang», es decir, que durante un período largo de tiempo no se tenga nada y de repente haya una situación completamente nueva. Por otra parte, también se logra que el usuario se implique estrechamente en la planificación de los pasos siguientes.

El modelo de desarrollo incremental también se utiliza para evitar la demanda de funcionalidades excesivas al sistema por parte de los usuarios, ya que como a éstos les resulta difícil definir sus necesidades reales tienden a pedir demasiado. Actuando con este modelo se atiende primero a las funcionalidades esenciales y las funcionalidades accesorias sólo se incluyen en las versiones sucesivas cuando realmente son necesarias. De esta manera no se corre el riesgo de emplear un esfuerzo considerable en implantar funcionalidades que realmente no son necesarias.

La diferencia entre el modelo de desarrollo evolutivo y el modelo de desarrollo incremental radica en dos aspectos:

- El modelo incremental parte de la hipótesis de que se conocen todos los requisitos y éstos se van incorporando al sistema en versiones sucesivas. En el modelo evolutivo sólo se conocen unos pocos requisitos y los restantes se van descubriendo en sucesivas evoluciones del prototipo.
- Cada vez que se desarrolla una nueva versión, en el modelo evolutivo es una versión de todo el sistema, mientras que en el incremental es una versión anterior sin cambios, más un número de nuevas funciones.

8. EL MODELO DE CICLO DE VIDA EN ESPIRAL.

Como alternativa a los modelos de ciclo de vida «tradicionales», esto es, el modelo en cascada y los modelos de prototipado, y especialmente para solventar los principales problemas del ciclo de vida clásico, Böehm, recogiendo la experiencia adquirida a lo largo de varios años de aplicar sucesivos refinamientos en el desarrollo de grandes proyectos, presentó en 1986 el modelo de ciclo de vida en espiral.

El modelo en espiral, al incorporar como nuevo elemento el análisis de riesgos, supone una visión más general del proceso de desarrollo del software que los otros modelos y puede ser aplicado a un amplio rango de sectores. Sin embargo, no es un modelo de ciclo de vida tan general como para que no tenga que estar sometido a unos métodos, disciplinas y restricciones sobre el camino de desarrollo.

Las principales diferencias entre el modelo en espiral y los modelos de ciclo de vida más tradicionales son las siguientes:

- En el modelo en espiral hay un reconocimiento explícito de las diferentes alternativas para alcanzar los objetivos del proyecto.
- El modelo en espiral se centra en la identificación de los riesgos asociados a cada alternativa y en la manera de resolver dichos riesgos.
- En el modelo en espiral los proyectos se dividen en ciclos (ciclos de espiral), avanzándose en el desarrollo mediante consensos al final de cada ciclo, en el sentido de que deberá existir un acuerdo para realizar cambios, o para dar por concluido el ciclo y pasar al siguiente, o para terminar el proyecto a la vista de lo que se haya aprendido desde el inicio del mismo.
- El modelo en espiral se adapta a cualquier tipo de actividad (p. ej., consulta de asesores expertos) necesaria para la consecución de los objetivos del proyecto.

El modelo en espiral refleja la idea de que cada ciclo implica una progresión en el desarrollo del producto software que aplica la misma secuencia de pasos para cada parte del producto y para cada uno de sus niveles de elaboración, desde la concepción global hasta la codificación individual de cada programa. Esta secuencia de pasos, iterativa en cada fase del desarrollo, se compone de las cuatro actividades siguientes, que se detallarán más adelante:

1. Planificación. Identificación de los objetivos, alternativas y restricciones de la fase.
2. Análisis del riesgo. Análisis de alternativas e identificación y resolución de los riesgos.
3. Ingeniería. Desarrollo del producto correspondiente a la fase en cuestión.
4. Evaluación del cliente. Valoración de los resultados de la ingeniería y planificación de la siguiente fase.

Según esto, el modelo se puede representar mediante unos ciclos externos de espiral, que representan las fases en que se ha dividido el desarrollo del proyecto software, normalmente las del modelo clásico, y unos ciclos internos, iterativos para cada fase, en los que se llevan a cabo las cuatro actividades antes citadas. La dimensión radial (véase figura 8) indica los costes de desarrollo acumulativos, mientras que la dimensión angular indica el progreso hecho en cumplimentar cada fase.

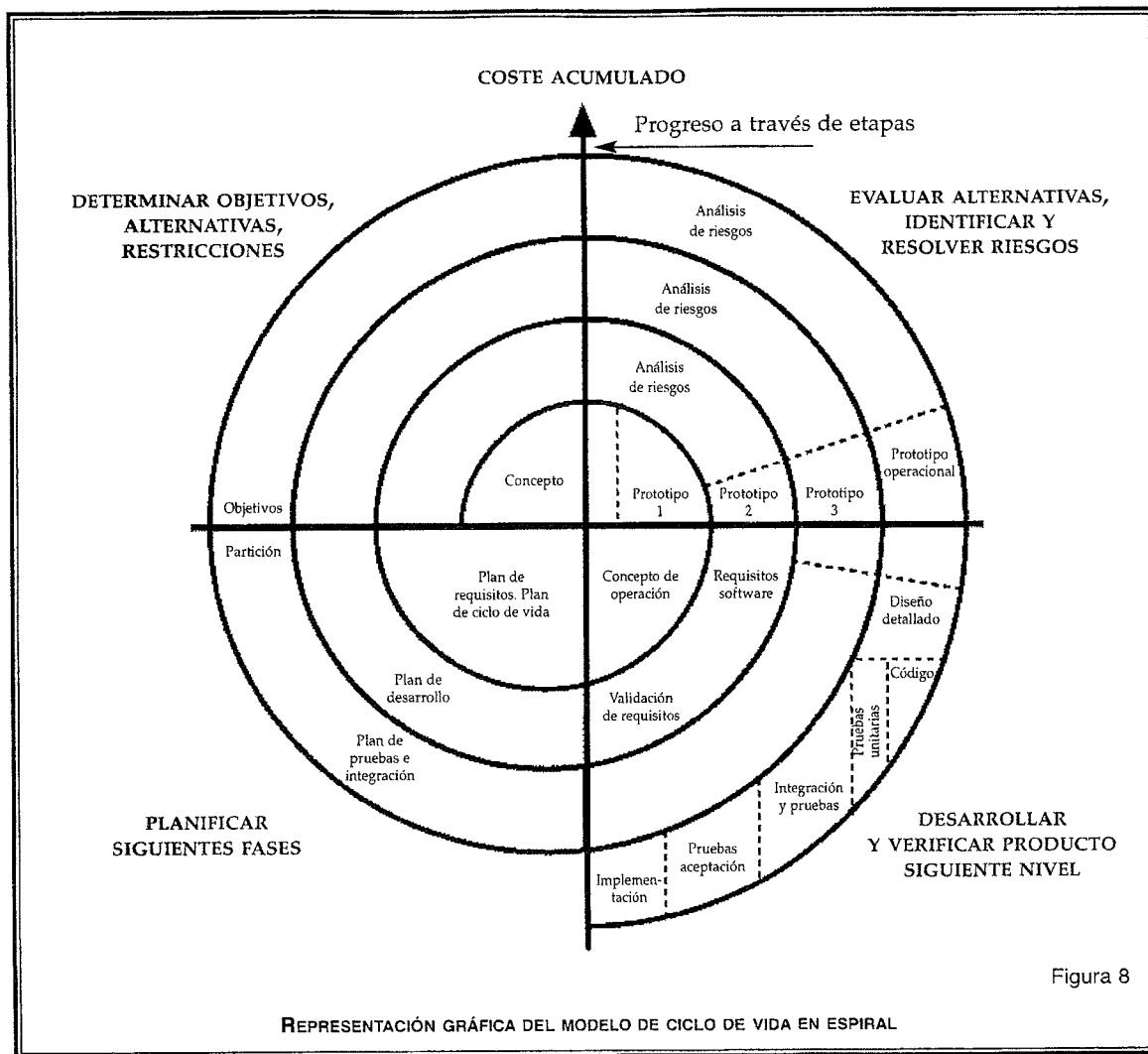


Figura 8

8.1. OPERATIVA DEL MODELO EN ESPIRAL.

Para aplicar el modelo de ciclo de vida en espiral al desarrollo de un proyecto software, el proyecto se divide en una serie de fases, que constituyen los ciclos externos de la espiral. Por ejemplo, estos ciclos externos, o en general, ciclos de espiral, pueden coincidir con las fases del modelo de ciclo de vida clásico.

Una vez determinados los ciclos de espiral, la operativa que sigue el modelo para cada uno de ellos es la siguiente:

1. Planificación. Este primer paso con el que comienza cada ciclo de espiral consiste en la determinación de:
 - Los objetivos de la parte del producto que está siendo elaborada (funcionalidad, rendimiento, adaptación a los cambios, etc.).
 - Las alternativas principales para realizar o implementar esta parte del producto (diseño A, diseño B, compra del software ya desarrollado, reutilización de un software ya existente, etc.).
 - Las restricciones impuestas (coste, plazo de realización, interfaces, etc.).

2. **Análisis de riesgos.** Es el paso más significativo del modelo en espiral. Comienza con la evaluación de cada alternativa respecto a los objetivos y a las restricciones. Este proceso de evaluación identificará áreas de incertidumbre que son fuentes significativas de riesgo en el proyecto, por lo que, seguidamente, se formulará una estrategia económicamente efectiva para resolver las fuentes de riesgo. Tal estrategia puede consistir en la realización de prototipos, aplicar técnicas de simulación, pasar cuestionarios a los usuarios y, en general, aplicar técnicas de resolución de riesgos.

Al hablar de riesgos se está entendiendo la posibilidad de que algo vaya mal. Por ejemplo, si se pretende utilizar un nuevo lenguaje de programación, un riesgo sería que no se dispusiese del compilador adecuado.

Los riesgos o incertidumbres que se identifiquen en cada fase pueden afectar tanto a aspectos conceptuales como externos del sistema. Son de tipo conceptual los que afectan al desarrollo, por ejemplo, que ni usuarios ni analistas tengan un entendimiento suficiente del problema, pobre capacidad de comunicación entre usuarios y analistas, etc.; y son de aspecto externo, básicamente, los que afectan al rendimiento del sistema.

Una vez identificados y evaluados los riesgos, la siguiente actuación vendrá determinada por la importancia relativa de los riesgos identificados, pudiéndose dar las siguientes situaciones:

- a) Predominan los riesgos de rendimiento o los de interfaces de usuario, sobre los riesgos de desarrollo. Ante esta situación, se procederá al desarrollo de prototipos evolutivos, pudiendo suceder que:
 1. El prototipo a que se llega es lo suficientemente operacional como para servir de base de bajo riesgo para la evolución futura del producto. Si es así, los pasos siguientes (ingeniería) consistirán en una serie continua de evolución de prototipos.
 2. El prototipo a que se llega resuelve todos los riesgos de rendimiento y de interfaces de usuario, pero no los riesgos de desarrollo. En este caso, en el paso siguiente (ingeniería) se habrá de seguir el modelo de ciclo de vida clásico, si bien modificado para incorporar el desarrollo incremental.
 - b) Predominan los riesgos de desarrollo. En este caso estamos en la misma situación que la expuesta para el caso a) 2.
3. **Ingeniería.** Este paso consiste en el desarrollo y verificación del producto objeto de la fase (ciclo de espiral) en que nos encontremos. Como esta implementación está dirigida por el riesgo, el desarrollo podrá seguir las pautas de un prototipado evolutivo, las del ciclo de vida clásico, las orientadas a transformaciones automáticas, o cualquier otro enfoque del desarrollo. En definitiva, esto permite al modelo en espiral acomodarse a cualquier mezcla de estrategias de desarrollo, las cuales se escogerán considerando la magnitud relativa de los riesgos y la efectividad relativa de las distintas alternativas para resolver estos riesgos.
 4. **Evaluación del cliente.** Una característica importante del modelo en espiral es que cada ciclo de espiral se completa con una revisión en la que participan aquellos que tienen relación con el producto (desarrolladores, usuarios, etc.). Esta revisión incluye todos los productos desarrollados durante el ciclo, los planes para el siguiente ciclo y los recursos necesarios para llevarlos a cabo.

Según el modelo de ciclo de vida en espiral, el desarrollo de un proyecto software debe comprender una serie de fases o ciclos (ciclos externos o ciclos de espiral), realizándose en cada uno de dichos ciclos las siguientes actividades:

PLANIFICACIÓN:

1. Definición de los objetivos del ciclo de espiral.
2. Identificación de las alternativas para implementar el producto asociado al ciclo de espiral.
3. Identificación de las restricciones impuestas (costes, calendario, etc.).

ANÁLISIS DE RIESGOS:

4. Evaluar cada alternativa en base a los objetivos y restricciones (→ identificar fuentes de riesgo) y elegir la más atractiva.
5. Decidir cómo resolver los riesgos asociados a la alternativa elegida.
6. Resolución de los riesgos. (La técnica a aplicar dependerá del tipo de riesgo dominante).

INGENIERÍA:

7. Desarrollar y verificar el producto objeto de la fase o ciclo de espiral en que nos encontremos. El desarrollo se hará en función del riesgo predominante (prototipado evolutivo, ciclo de vida clásico, etc.).

EVALUACIÓN POR EL CLIENTE:

8. Analizar y evaluar los resultados (→ consensuar el paso al siguiente ciclo).
9. Planificar el siguiente ciclo de espiral.

RESUMEN CICLO DE VIDA EN ESPIRAL

8.2. VENTAJAS E INCONVENIENTES DEL MODELO EN ESPIRAL.

Para concluir el estudio del modelo de ciclo de vida en espiral haremos una somera referencia a los pros y los contras que presenta su utilización.

La principal ventaja del modelo en espiral es el amplio rango de opciones a que puede ajustarse y que éstas permiten utilizar los modelos de proceso de construcción de software tradicionales; por otra parte, su orientación al riesgo evita, si no elimina, muchas de las posibles dificultades.

Otras ventajas adicionales son:

- Concentra su atención en opciones que permiten la reutilización de software ya existente.
- Se centra en la eliminación de errores y alternativas poco atractivas.
- No establece procedimientos diferentes para el desarrollo del software y el mantenimiento del mismo.
- Proporciona un marco estable para desarrollos integrados hardware-software.
- Permite preparar la evolución del ciclo de vida del producto software, así como el crecimiento y cambios de éste.
- Permite incorporar objetivos de calidad en el desarrollo de productos software.
- Para cada fuente de actividad del proyecto y gasto de recursos se pregunta ¿cuánto es suficiente?, es decir, ¿cuántos requerimientos de planificación, análisis, garantía de calidad, pruebas, etc., debe incluir el proyecto? Utilizando un método orientado a riesgos se puede determinar el nivel de esfuerzo necesario en función del nivel de riesgo incurrido por no hacer lo suficiente.
- Se adapta muy bien al diseño y programación orientada a objetos. Posiblemente con este método es cuando obtiene mejores resultados.

En cuanto a los inconvenientes que plantea la utilización del modelo en espiral, cabe citar:

- Dificultad para adaptar su aplicabilidad al software contratado, debido a la poca flexibilidad y libertad de éste.
- Dependencia excesiva de la experiencia que se tenga en la identificación y evaluación de riesgos.
- Necesidad de una elaboración adicional de los pasos del modelo, lo que depende también, en gran medida, de la experiencia del personal.

9. MODELOS BASADOS EN TRANSFORMACIONES.

Dentro de los modelos de ciclo de vida «no tradicionales», además del modelo en espiral, hay que hacer mención a un conjunto de modelos de desarrollo del software, que genéricamente vamos a denominar «modelos basados en transformaciones» y cuyo punto en común es que se basan en la utilización de diversas herramientas.

Estos modelos, también llamados «ciclo de vida con producción automática de diseño y código», se han propuesto como solución al problema que plantean los modelos de desarrollo evolutivo de producir software mal estructurado. Se basan en la posibilidad de convertir automáticamente una especificación formal de un producto software en un programa que satisfaga las especificaciones, utilizando herramientas de cuarta generación. Para ello, los pasos típicos que siguen estos modelos son:

1. Especificación formal del producto tal como lo permita la comprensión inicial del problema.
2. Transformación automática de la especificación en código.

3. Realizar bucles iterativos para mejorar el rendimiento del código resultante.
4. Probar el producto resultante.
5. Reajustar las especificaciones para dejarlas en concordancia con el resultado de la experiencia operativa y volver a generar el código a partir de las especificaciones, volviendo a optimizar y probar el producto.

La dificultad que presentan estos modelos es que las posibilidades de transformación automática generalmente sólo están disponibles para productos relativamente pequeños y aplicados a unas áreas muy limitadas.

De todos los modelos basados en transformaciones tal vez los más conocidos son el modelo basado en técnicas de cuarta generación propuesto por Pressman como uno de los paradigmas para el desarrollo del software y el modelo de transformación propuesto por Carma McClure.

9.1. EL MODELO DE TÉCNICAS DE CUARTA GENERACIÓN.

El término «técnicas de cuarta generación» (T4G) engloba un amplio conjunto de herramientas de software que tienen en común el facilitar la especificación de algunas de las características del software a alto nivel y a partir de la misma generar el código fuente. Dicho de otra forma, estas técnicas persiguen la especificación del software a un nivel más próximo al lenguaje natural o en una notación que proporcione funciones significativas.

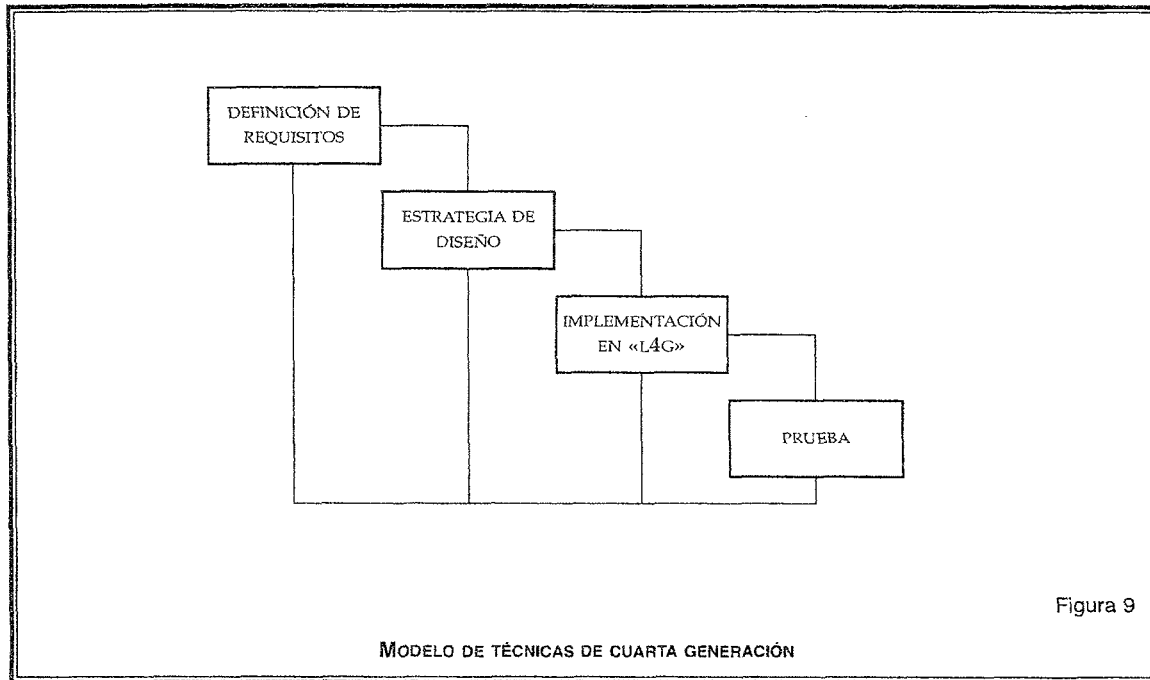
Un entorno de T4G debe incluir todas o algunas de las siguientes herramientas:

- Lenguajes no procedimentales para consultas a bases de datos.
- Generadores de código.
- Generadores de pantallas.
- Generadores de informes.
- Herramientas de manipulación de datos.
- Facilidades gráficas de alto nivel.

El modelo de desarrollo comienza con la fase de definición de requisitos. En un caso ideal, los requisitos definidos por el usuario serían directamente traducidos a un prototipo operativo; sin embargo, la realidad suele ser bastante diferente ya que, por lo general, el usuario puede no estar seguro de lo que necesita, puede ser ambiguo en la especificación de hechos que le son conocidos y puede no saber cómo expresar sus necesidades en la forma en que una herramienta de cuarta generación la necesita. Ante ello, se hace imprescindible el diálogo entre usuario y analista a fin de definir perfectamente los requisitos y determinar una estrategia de diseño del software.

Determinados los requisitos y la estrategia de diseño, la utilización de un lenguaje de cuarta generación no procedimental (L4G) permitirá que la representación de los resultados deseados se traduzca automáticamente en el código fuente que produce esos resultados y conducirá a la implementación del sistema.

Evidentemente, antes de que una implementación con técnicas de cuarta generación se convierta en el sistema final se deben realizar pruebas completas, producir la documentación adecuada y verificar que el software desarrollado es fácilmente mantenible.



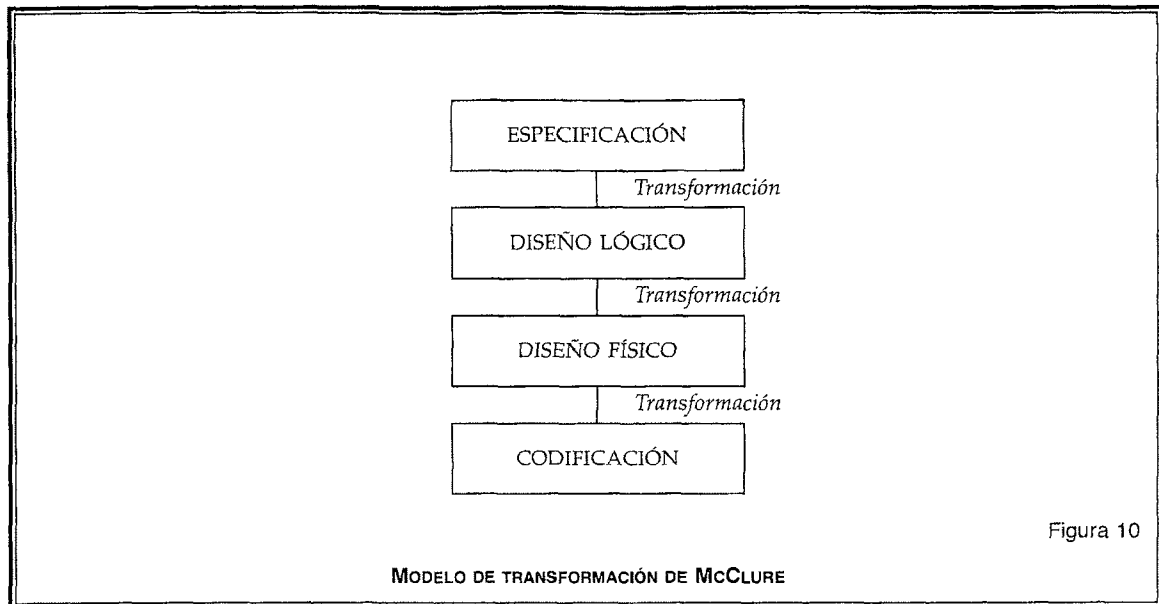
9.2. EL MODELO DE TRANSFORMACIÓN DE McCLURE.

El modelo de transformación propuesto por Carma McClure se basa en la utilización de herramientas de ingeniería asistida por ordenador (herramientas CASE -Computer Aided Software Engineering-) y siguiendo el ciclo de vida clásico ir transformando, mediante el uso de estas herramientas, las especificaciones funcionales en el diseño lógico, éste en el diseño físico y éste a su vez en el código fuente correspondiente.

El ciclo de vida del software puede ser considerado como una serie sucesiva de transformaciones, las cuales, gracias a la utilización de herramientas CASE, se producirán, en su mayor parte, de forma automática.

Las ventajas que introduce el uso de la tecnología CASE en el desarrollo del software son, entre otras, las siguientes:

- Comprobación de errores en las etapas iniciales del desarrollo, ya que estas herramientas permiten realizar análisis de completud y consistencia de los requisitos.
- Posibilidad de realizar el mantenimiento en la especificación, puesto que con los generadores de código se puede mantener los programas mediante la modificación de objetos de nivel de abstracción de diseño y de análisis.
- Reusabilidad de objetos de desarrollo (diseños, especificaciones, programas, etc.).
- Potencia la especificación orientada al problema en vez de a la máquina o al lenguaje utilizado.



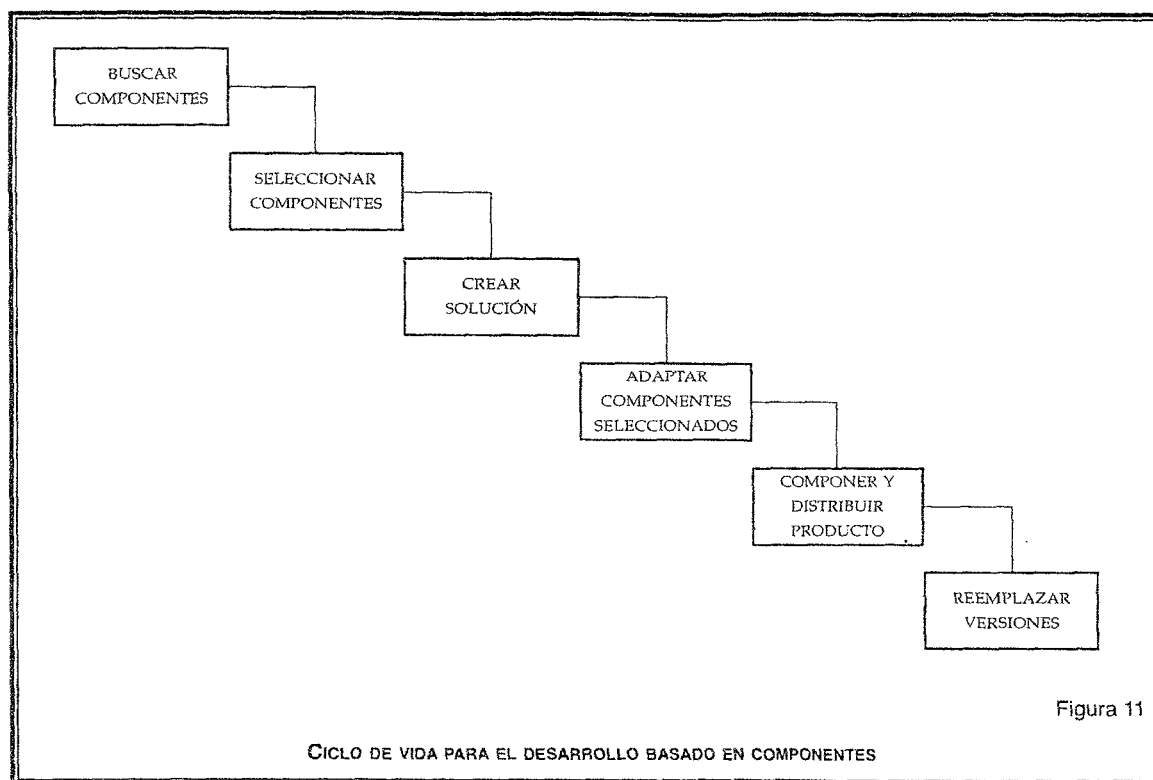
10. CICLO DE VIDA CON REUTILIZACIÓN.

En consonancia con las nuevas tecnologías de desarrollo del software, tales como el Desarrollo de Software Basado en Componentes (DSBC), ha surgido recientemente un nuevo modelo de ciclo de vida basado en la reutilización cuyo enfoque básico es configurar y especializar componentes de software ya existentes.

En estos modelos de ciclo de vida basados en el ensamblaje de componentes, el tamaño, complejidad y capacidad funcional de los componentes, es decir, su granularidad, varía mucho según los diferentes enfoques. La mayoría de los modelos intentan utilizar componentes de grano fino, esto es, componentes similares a estructuras de datos con los algoritmos incorporados para su manipulación, si bien esto no constituye un enfoque distinto al desarrollo de software tradicional. Otros enfoques intentan utilizar componentes ensamblando funcionalmente sistemas o subsistemas completos (componentes de grano grueso), por ejemplo, sistemas de gestión de interfaces de usuario. En este caso sí estamos ante un enfoque alternativo al desarrollo de sistemas software.

En cualquier caso, los pasos de que consta el ciclo de desarrollo para un sistema basado en componentes, lo que genéricamente estamos denominando ciclo de vida con reutilización, son:

1. Buscar componentes, tanto COTS (Comercial Off-The-Shelf) como no COTS.
2. Seleccionar los componentes más adecuados para el sistema.
3. Crear una solución compuesta que integre la solución previa.
4. Adaptar los componentes seleccionados de forma que se ajusten al modelo de componentes o a los requisitos de la aplicación.
5. Componer y distribuir el producto.
6. Reemplazar versiones anteriores o mantener las partes COTS y no COTS del sistema.



Los dos primeros pasos, buscar y seleccionar componentes, son los más importantes del ciclo ya que se deberán tomar decisiones a largo plazo sobre qué componentes serán empleados en el sistema. Usar componentes que no sean adecuados para el desarrollo del sistema puede convertirse en algo muy costoso para la empresa.

Por otra parte, además de los problemas inherentes a la reutilización del software, los productos COTS presentan problemas específicos como incompatibilidad, inflexibilidad, y complejidad, por lo que el establecimiento de métodos sistemáticos y repetibles para evaluar y seleccionar dichos componentes es un aspecto importante para el desarrollo del software basado en componentes y, en general, para la Ingeniería del Software Basada en Componentes (ISBC).

A diferencia del prototipado, que intenta reducir los costes de desarrollo mediante implementaciones parciales, la reutilización de componentes software pretende reducir estos costes incorporando en productos software nuevos diseños y código previamente probados.

11. GESTIÓN DEL CICLO DE VIDA DEL SOFTWARE.

Al estudiar el concepto de ciclo de vida indicábamos que entre las características que debe reunir cualquier modelo de ciclo de vida se encuentran la de establecer los criterios de transición para pasar de una fase a la siguiente y la de establecer puntos de control para la gestión del proyecto. Dicho de otra manera, el resultado de utilizar un modelo de ciclo de vida tiene que ser un proceso controlable.

Dada la naturaleza intangible del software, la gestión de los procesos de desarrollo generalmente se basa en la adopción de modelos que hagan visibles esos procesos mediante documentos, informes o revisiones. Son los modelos «orientados a entregas de documentos», en los que el proceso de desarrollo se divide en etapas y no se considera terminada una etapa hasta que el documento resultado de la misma ha si-

do producido, revisado y aceptado, puesto que este documento sirve de entrada de definición para la siguiente fase. El ejemplo más característico de estos modelos es, lógicamente, el modelo de ciclo de vida en cascada y de hecho suelen estar definidos los productos a entregar al final de cada fase.

Sin embargo, aunque los modelos «orientados a entregas de documentos» son los más utilizados puesto que son los que más facilitan la gestión y control del proceso de desarrollo del software al realizarse éste sobre el propio documento, presentan algunos inconvenientes tales como:

- No coincidencia en el tiempo del momento en que la dirección necesita un documento para ver el progreso del proyecto y el momento de terminación de una fase, lo que puede dar lugar a que se produzcan documentos «artificiales».
- Se requiere un tiempo significativo para revisar y aprobar los documentos, lo que hace que la transición de una fase a la siguiente no sea lo suficientemente suave.
- El que el documento de salida de una fase sea la entrada para la siguiente presupone que el proceso de desarrollo es lineal, lo que no es aplicable a muchos proyectos. Además, hay que tener en cuenta que en cualquier fase del proyecto los ingenieros de software tienen en consideración tanto los pasos anteriores como los futuros.

En los modelos no orientados a documentos, precisamente por resultar muy cara la generación de documentación, la gestión del ciclo de vida habrá de realizarse sobre otro tipo de «entregables» que se produzcan. Por ejemplo, en el modelo de evolución de prototipos el control del proyecto se realizará sobre cada uno de los prototipos que se vayan produciendo, fundamentalmente mediante la realización de pruebas.

En cualquier caso, es importante resaltar que la gestión del ciclo de vida del software es imprescindible pues es lo que asegura la calidad del desarrollo del mismo, independientemente de que este control de calidad se realice sobre documentos, mediante revisiones o inspecciones detalladas de los mismos, o sobre cualquier otro tipo de «entregable», mediante distintos tipos de pruebas.

BIBLIOGRAFÍA

- Ingeniería del Software. Un enfoque práctico. Roger S. Pressman. Ed. McGraw Hill.
- Ingeniería del Software. Ian Sommerville. Ed. Addison-Wesley.
- Ingeniería del Software de Gestión. Antonio de Amescua, Paloma Martínez y otros. Ed. Paraninfo.
- CASE. La automatización del Software. Carma McClure. Ed. Ra-Ma.
- Temario de las pruebas selectivas para ingreso en el Cuerpo Superior de Sistemas y Tecnologías de la Información de la Administración del Estado. ASTIC.
- Temario de las pruebas selectivas para el acceso, por promoción interna, al Cuerpo de Gestión de Sistemas e Informática de la Administración del Estado. Ministerio para las Administraciones Públicas.
- Temario del Máster en Ingeniería del Software. Facultad de Informática. Universidad Politécnica de Madrid. Ed. Centro de Estudios Financieros.



