TEMA

48

CEDE

INFORMÁTICA

Desarrollo de los temas

Ingeniería del «software». Ciclo de desarrollo del «software». Tipos de ciclos de desarrollo. Metodologías de desarrollo. Características distintivas de las principales metodologías de desarrollo utilizadas en la Unión Europea.

elaborado por
EL EQUIPO DE PROFESORES
DEL CENTRO DOCUMENTACIÓN

GUIÓN - ÍNDICE

1. INGENIERÍA DEL SOFTWARE

2. METODOLOGÍAS DE DESARROLLO SOFTWARE

2.1. Metodologías de desarrollo tradicional
- 2.1.1. Ciclo de vida de desarrollo software
- 2.1.2. Elementos del ciclo de vida software
- 2.1.3. Tipos del ciclo de vida en el desarrollo software
  - 2.1.3.1. Ciclo de vida clásico en cascada o clásico
  - 2.1.3.2. Ciclo con técnicas de cuarta generación
  - 2.1.3.3. Construcción de prototipos
  - 2.1.3.4. Ciclo de vida en espiral
  - 2.1.3.5. Desarrollo por etapas
  - 2.1.3.6. XP o Programación extrema
  - 2.1.3.7. Modelo iterativo o creciente

3. CARACTERÍSTICAS DISTINTIVAS DE LAS PRINCIPALES METODOLOGÍAS DE DESARROLLO UTILIZADAS EN LA UNIÓN EUROPEA

3.1. MÉTRICA V3
3.2. Seis Sigma
3.3. CMMI

BIBLIOGRAFÍA

- **Análisis y Diseño de Sistemas**. Editorial McGraw-Hill.

- **Ingeniería del Software**. Editorial McGraw-Hill.

- GAIL, Linda, CHRISTIE, John. **Enciclopedia de Términos de Computación**. Editorial PHH, Pentice Hall.

- PHAN THU. **Merise Apliquée**. Eyrolles.

- MATHERON, Merise. **Metodología de desarrollo de Sistemas**. Paraninfo.

- FISHER, A. **C.A.S.E. Using Software Development Tools**. Wiley.

- WILSON, P. **Computer Supported Cooperative Work**. Intellect Oxford.

- **MAP, Metodología de planificaión de S.I. bajo metodología Métrica Versión 3, Guía de Referencia, Guía de Técnicas**. Ministerio.

1. INGENIERÍA DEL SOFTWARE

El desarrollo de sistemas de software complejos no es una actividad trivial, que pueda abordarse sin un análisis previo. El considerar que un proyecto de desarrollo de software podía abordarse como cualquier otro, ha llevado a una serie de problemas que limitan nuestra capacidad de aprovechar los recursos que el hardware pone a nuestra disposición.

No existe una fórmula mágica para solucionar estos problemas, pero combinando métodos aplicables a cada una de las fases del desarrollo de software, construyendo herramientas para automatizar estos métodos, utilizando técnicas para garantizar la calidad de los productos desarrollados y coordinando todas las personas involucradas en el desarrollo de un proyecto, podremos avanzar mucho en la solución de estos. De ello se encarga la disciplina llamada *Ingeniería del Software*.

Una de las primeras definiciones que se dio de la ingeniería del software es la siguiente:

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

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

Los **métodos** indican cómo construir técnicamente el software, y abarcan una amplia serie de tareas que incluyen la planificación y estimación de proyectos, el análisis de requisitos, el diseño de estructuras de datos, programas y procedimientos, la codificación, las pruebas y el mantenimiento. Los métodos introducen frecuentemente una notación específica para la tarea en cuestión y una serie de criterios de calidad.

Las **herramientas** proporcionan un soporte automático o semiautomático para utilizar los métodos. Existen herramientas automatizadas para cada una de las fases vistas anteriormente, y sistemas que integran las herramientas de cada fase de forma que sirven para todo el proceso de desarrollo. Estas herramientas se denominan CASE (Computer Assisted Software Engineering).

Los **procedimientos** definen la secuencia en que se aplican los métodos, los documentos que se requieren, los controles que permiten asegurar la calidad y las directrices que permiten a los gestores evaluar los progresos.

2. METODOLOGÍA DE DESARROLLO SOFTWARE

Se puede definir **método** como la ordenación y explicitación formal de los medios que conducen eficazmente al logro de un objetivo predeterminado ó de una tarea concreta.

A partir de esta definición, se puede decir que una **metodología** es el conjunto de métodos que se usan en una determinada actividad con el fin de formalizarla y optimizarla.

Existen claramente dos tipos de metodologías software:

- Metodologías de desarrollo de software tradicional.
- Metodologías de desarrollo ágiles.

2.1. METODOLOGÍAS DE DESARROLLO TRADICIONALES

2.1.2. Ciclo de vida del desarrollo software

Todo proyecto de ingeniería tiene unos fines ligados a la obtención de un producto, proceso o servicio que es necesario generar a través de diversas actividades. Algunas de estas actividades pueden agruparse en fases porque globalmente contribuyen a obtener un producto intermedio, necesario para continuar hacia el producto final y facilitar la gestión del proyecto.

Al conjunto de las fases empleadas se le denomina "**ciclo de vida**".

Sin embargo, la forma de agrupar las actividades, los objetivos de cada fase, los tipos de productos intermedios que se generan, etc. Pueden ser muy diferentes dependiendo del tipo de producto o proceso a generar y de las tecnologías empleadas.

La complejidad de las relaciones entre las distintas actividades crece exponencialmente con el tamaño, con lo que rápidamente se haría inabordable si no fuera por la vieja táctica de "divide y vencerás". De esta forma la división de los proyectos en fases sucesivas es un primer paso para la reducción de su complejidad, tratándose de escoger las partes de manera que sus relaciones entre sí sean lo más simples posibles.

La definición de un ciclo de vida facilita el **control sobre los tiempos** en que es necesario aplicar recursos de todo tipo (personal, equipos, suministros, etc.) al proyecto.

El **control de calidad** también se ve facilitado si la separación entre fases se hace corresponder con puntos en los que ésta deba verificarse (mediante comprobaciones sobre los productos parciales obtenidos).

Por ciclo de vida referido al desarrollo software, 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 tipos de ciclo de vida, es decir, diversas formas de ver el proceso de desarrollo de software, y cada uno de ellos va asociado a un paradigma de la ingeniería del software, es decir, a una serie de métodos, herramientas y procedimientos que debemos usar a lo largo de un proyecto.

La elección de un paradigma 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.

2.1.2. Elementos del ciclo de vida software

Un ciclo de vida para un proyecto software se compone de **fases sucesivas** compuestas por tareas planificables.

Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con **bucles de realimentación**, de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo en cada pasada de ejecución aportaciones de los resultados intermedios que se van produciendo (realimentación).

Para un adecuado control de la progresión de las fases de un proyecto se hace necesario especificar con suficiente precisión los resultados evaluables, o sea, productos intermedios, en este caso software con determinadas funcionalidades o requisitos, que deben resultar de las tareas incluidas en cada fase. Normalmente estos productos marcan los hitos entre fases.

A continuación presentamos los distintos elementos que integran un ciclo de vida:

Fases. Una fase es un conjunto de actividades relacionadas con un objetivo en el desarrollo software del proyecto. Se construye agrupando tareas (actividades elementales) que pueden compartir un tramo determinado del tiempo de vida de un proyecto. La agrupación temporal de tareas impone requisitos software temporales correspondientes a la asignación de recursos (humanos, financieros o materiales).

Cuanto más grande y complejo sea un proyecto, mayor detalle se necesitará en la definición de las fases para que el contenido de cada una siga siendo manejable. De esta forma, cada fase de un proyecto puede considerarse un "*micro-proyecto*" en sí mismo, compuesto por un conjunto de micro-fases.

Cada fase viene definida por un conjunto de elementos observables externamente, como son las **actividades** con las que se relaciona, los **datos de entrada** (resultados de la fase anterior, documentos o productos requeridos para la fase, experiencias de proyectos anteriores), los **datos de salida** (resultados a utilizar por la fase posterior, experiencia acumulada, pruebas o resultados efectuados) y la **estructura interna** de la fase.

Entregables ("*deliverables*"). Son los productos intermedios que generan las fases. Para un proyecto software, estos entregables pueden ser materiales (componentes, equipos) o inmateriales (documentos, software). Los entregables permiten evaluar la marcha del proyecto mediante comprobaciones de su adecuación o no a los requisitos funcionales y de condiciones de realización previamente establecidos. Cada una de estas evaluaciones puede servir, además, para la toma de decisiones a lo largo del desarrollo del proyecto.

2.1.3. Tipos de ciclo de vida en el desarrollo software

Las principales diferencias entre distintos modelos de ciclo de vida están en:

- El **alcance** del ciclo dependiendo de hasta dónde llegue el proyecto correspondiente. Un proyecto software puede comprender un simple estudio de viabilidad del desarrollo de un producto, o su desarrollo completo o, la evolución del producto con su desarrollo, fabricación, y modificaciones posteriores hasta su retirada del mercado.

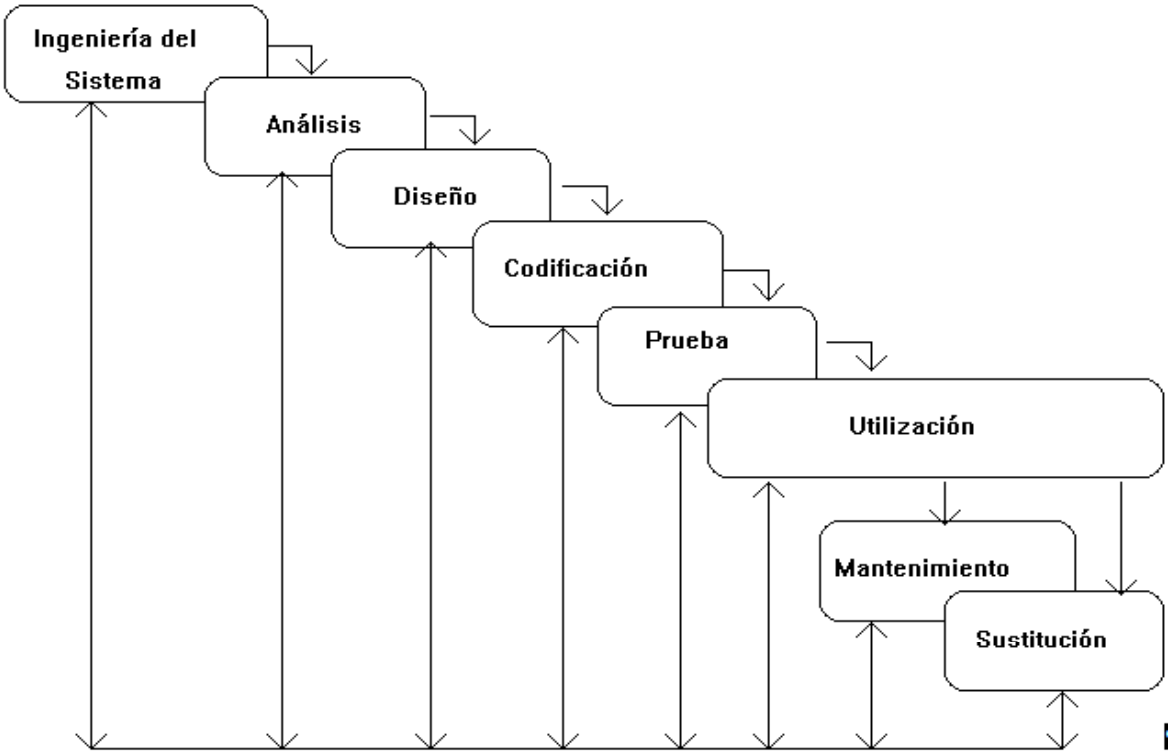
- Las **características** (contenidos) de las fases en que dividen el ciclo del desarrollo software o de la **organización** (interés de reflejar en la división en fases aspectos de la división interna o externa del trabajo).

- La **estructura** de la sucesión de las fases que puede ser de diferentes formas, tal y como se describe en los siguientes apartados.

2.1.3.1. Ciclo de vida en cascada o clásico

Este paradigma es el más antiguo de los empleados en la IS 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.



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:

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 comporta-

miento del sistema del cual el software va a formar parte. En el caso de que queramos construir un sistema nuevo, por ejemplo un sistema de control, deberemos analizar cuáles son los requisitos y la función del sistema, y luego se asignarán un subconjunto de estos requisitos al software. En el caso de un sistema ya existente (supongamos, por ejemplo, que queremos informatizar una empresa) se debe analizar el funcionamiento de la misma –las operaciones que se llevan a cabo en ella–, y asignaremos al software aquellas funciones que vamos a automatizar.

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.

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 las 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.

Diseño

El diseño se aplica a cuatro características distintas del software: la estructura de los datos, la arquitectura de las aplicaciones, la estructura interna de los programas y las interfaces.

El diseño es el proceso que traduce los requisitos en una representación del software de forma que pueda conocerse la arquitectura, funcionalidad e incluso la calidad del mismo 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).

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.

Podemos observar que estas primeras fases del ciclo de vida consisten básicamente en una **traducción**: en el análisis del sistema, los requisitos, la función y la estructura de este se traducen a un documento: el análisis del sistema que está formado en parte por diagramas y en parte por descripciones en lenguaje natural. En el análisis de requisitos se profundiza en el estudio del componente software del sistema y esto se traduce a un documento, también formado por diagramas y descripciones en lenguaje natural. En el diseño, los requisitos del software se traducen a una serie de diagramas que representan la estructura del sistema software, de sus datos, de sus programas y de sus interfaces. Por último, en la codificación se traducen estos diagramas de diseño a un lenguaje fuente, que luego se traduce –se compila– para obtener un programa ejecutable.

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.

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.

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

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, y también para los desarrolladores, 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.

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.

2.1.3.2. Ciclos con 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.

Al igual que en otros paradigmas, el proceso comienza con la recolección de requisitos, que pueden ser traducidos directamente a código fuente usando un generador de código. Sin embargo 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.

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.).

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.

Con respecto a las pruebas, podemos suponer 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 eficiente 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.

Como conclusión, podemos decir que, mediante el uso de técnicas de cuarta generación no se han obtenido (afortunadamente) los resultados previstos cuando estas herramientas

comenzaron a desarrollarse a principios de los ochenta (estos resultados incluían la desaparición de la codificación manual y con ello de los programadores, e incluso de los analistas, al poder encargarse el propio cliente 'con unos pequeños conocimientos técnicos' de manejar el generador), puesto que los avances en procesamiento de lenguaje natural (siempre ambiguo) no han sido (ni se espera que sean en un futuro próximo) demasiado grandes ni se han desarrollado lenguajes formales de especificación con la potencia expresiva necesaria.

Sin embargo, 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 (ej.: SAP).

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. 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. Esta crítica podría aplicarse a cualquier lenguaje de programación con respecto al ensamblador (los programas codificados en ensamblador siempre serán más rápidos y más pequeños que los generados por cualquier compilador), 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).

Sólo son aplicables al software de gestión. Esto es cierto, la mayoría de las herramientas de cuarta generación están orientadas a la generación de informes a partir de grandes bases de datos, pero últimamente están surgiendo herramientas que generan *esquemas de código* para aplicaciones de ingeniería y de tiempo real.

2.1.3.3. 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.

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 o al modelo en espiral. 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.

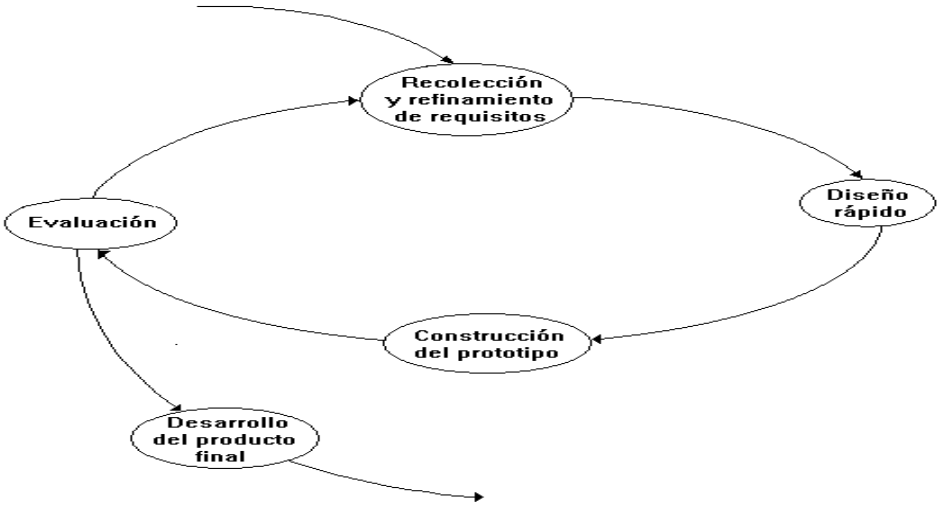
Según esto un prototipo puede tener alguna de las tres formas siguientes:

• En papel o ejecutable en ordenador, que describa la interacción hombre-máquina y los listados del sistema.

• 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 siguiente figura:



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.

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 funcionales de los programas: nos fijaremos más en la forma y en la apariencia que en el contenido.

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. La posibilidad más reciente consiste en el uso de especificaciones formales, que últimamente tienden al desarrollo de entornos interactivos (ej. OBJ) que faciliten el desarrollo incremental de especificaciones y permitan la prueba de estas especificaciones. En cualquier caso, el objetivo es siempre que la codificación sea rápida, aunque sea en detrimento de la calidad del software generado.

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.

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. Además, tras realizar varias vueltas en el ciclo de construcción de prototipos, el diseño del mismo se parece cada vez más al que tendrá el 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.

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.

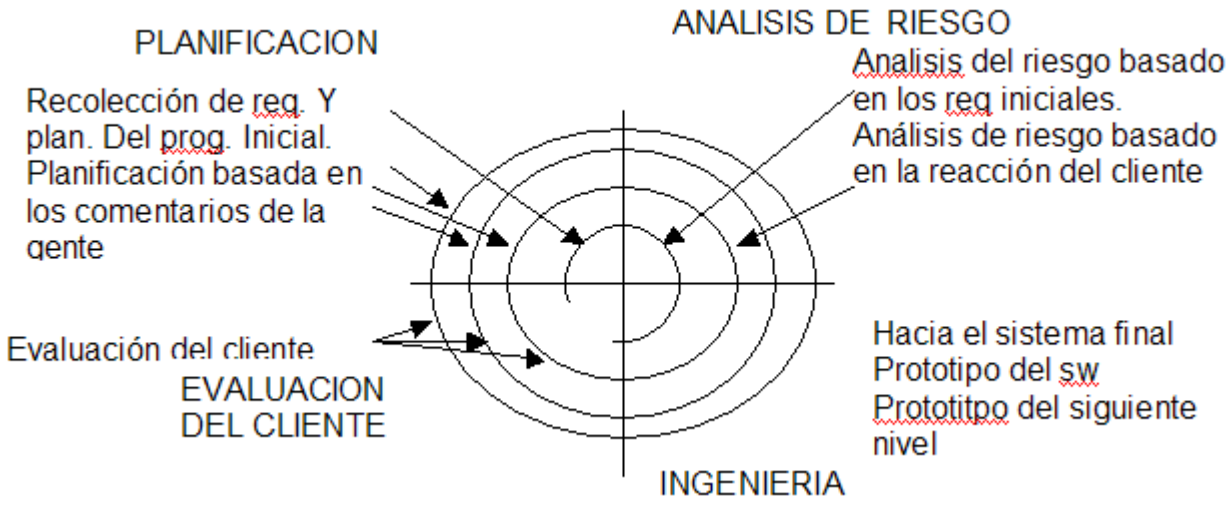
Uno de los problemas que suelen aparecer siguiendo el paradigma de construcción de prototipos, es que con demasiada frecuencia el prototipo pasa a ser parte del sistema final, bien sea por presiones del cliente, que quiere tener el sistema funcionando lo antes posible o bien porque los técnicos se han 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.

2.1.3.4. Ciclo de vida 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. Este es un modelo relativamente nuevo y no ha sido tan usado como los dos anteriores, aunque es de esperar que se extienda cada vez más.

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:

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.

Análisis de riesgo

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

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).

Ingeniería

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

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 una de las iteraciones se realiza el análisis de riesgos, teniendo en cuenta los requisitos y la reacción del cliente ante el último prototipo. Si los riesgos son demasiado grandes se terminará el proyecto, aunque lo normal es que se siga avanzando a lo largo de la espiral.

Con cada iteración, se construyen 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.

2.1.3.5. Desarrollo por etapas

El modelo de desarrollo de software por etapas es similar al Modelo de prototipos ya que se muestra al cliente el software en diferentes estados sucesivos de desarrollo, se diferencia en que las especificaciones no son conocidas en detalle al inicio del proyecto y por tanto se van desarrollando simultáneamente con las diferentes versiones del código.

Pueden distinguirse las siguientes fases:

- Especificación conceptual.
- Análisis de requerimientos.
- Diseño inicial.
- Diseño detallado, codificación, depuración y liberación.

Estas diferentes fases se van repitiendo en cada etapa del diseño.

2.1.3.6. XP o programación extrema

Se entiende como **Desarrollo ágil de software** a un paradigma de Desarrollo de Software basado en procesos ágiles. Los procesos ágiles de desarrollo de software, intentan evitar los burocráticos caminos de las metodologías tradicionales enfocándose en la gente y los resultados.

Es un marco de trabajo conceptual de la ingeniería de software que promueve iteraciones en el desarrollo a lo largo de todo el ciclo de vida del proyecto. Existen muchos métodos de desarrollo ágil; la mayoría minimiza riesgos desarrollando software en cortos lapsos de tiempo. El software desarrollado en una unidad de tiempo es llamado una iteración, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. Una iteración no debe agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, pero la meta es tener un demo (sin errores) al final de cada iteración. Al final de cada iteración el equipo vuelve a evaluar las prioridades del proyecto.

Los métodos Agiles enfatizan las comunicaciones cara a cara en vez de la documentación. La mayoría de los equipos “Agiles” están localizados en una simple oficina abierta, a veces llamadas "plataformas de lanzamiento" (bullpen en inglés).

El equipo “Agil” debe incluir revisores, diseñadores de iteración, escritores de documentación y ayuda y directores de proyecto. Los métodos ágiles también enfatizan que el software funcional es la primera medida del progreso. Combinado con la preferencia por las comunicaciones cara a cara, generalmente los métodos ágiles son criticados y tratados como "indisciplinados" por la falta de documentación técnica.

Aunque también existen otras metodologías ágiles como RUP o Scrum. XP es la que se expone en este tema por ser la mas exitosa.

La **programación extrema** o *eXtreme Programming* (XP) es un enfoque de la ingeniería de software formulado por Kent Beck, autor del primer libro sobre la materia, *Extreme Programming Explained: Embrace Change*. Es el más destacado de los procesos ágiles de desarrollo de software. Al igual que éstos, la programación extrema se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad.

Los defensores de XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos.

Ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto, y aplicarlo de manera dinámica durante el ciclo de vida del software.

Las características fundamentales del método son:

- **Desarrollo iterativo e incremental**: pequeñas mejoras, unas tras otras.

- **Pruebas unitarias continuas**, frecuentemente repetidas y automatizadas, incluyendo pruebas de regresión. Se aconseja escribir el código de la prueba antes de la codificación. Véase, por ejemplo, las herramientas de prueba JUnit orientada a Java, DUnit

orientada a Delphi y NUnit para la plataforma.NET. Estas dos últimas inspiradas en JUnit.

• **Programación en parejas**: se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto. Se supone que la mayor calidad del código escrito de esta manera, el código es revisado y discutido mientras se escribe, es más importante que la posible pérdida de productividad inmediata.

• Frecuente **integración del equipo de programación con el cliente** o usuario. Se recomienda que un representante del cliente trabaje junto al equipo de desarrollo.

• **Corrección de todos los errores** antes de añadir nueva funcionalidad. Hacer entregas frecuentes.

• **Refactorización del código**, es decir, reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo.

• **Propiedad del código compartida**: en vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve el que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados.

• **Simplicidad** en el código: es la mejor manera de que las cosas funcionen. Cuando todo funcione se podrá añadir funcionalidad si es necesario. La programación extrema apuesta que es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo.

• La simplicidad y la comunicación son extraordinariamente complementarias. Con más comunicación resulta más fácil identificar qué se debe y qué no se debe hacer. Mientras más simple es el sistema, menos tendrá que comunicar sobre este, lo que lleva a una comunicación más completa, especialmente si se puede reducir el equipo de programadores.

2.1.3.7. Modelo iterativo o creciente

Desarrollo iterativo y creciente (o incremental) es un proceso de desarrollo de software, creado en respuesta a las debilidades del modelo tradicional de cascada.

Para apoyar el desarrollo de proyectos por medio de este modelo se han creado frameworks (entornos de trabajo), de los cuales los dos más famosos son el Rational Unified Process (RUP) y el Dynamic Systems Development Method. El desarrollo incremental e iterativo es también una parte esencial de un tipo de programación conocido como Extreme Programming y los demás frameworks de desarrollo rápido de software.

Características

Usando análisis y mediciones como guías para el proceso de mejora es una diferencia mayor entre las mejoras iterativas y el desarrollo rápido de aplicaciones, principalmente por dos razones:

– Provee de soporte para determinar la efectividad de los procesos y de la calidad del producto.
– Permite estudiar y después mejorar y ajustar el proceso para el ambiente en particular.

Estas mediciones y actividades de análisis pueden ser añadidas a los métodos de desarrollo rápido existentes.

De hecho, el contexto de iteraciones múltiples conlleva ventajas en el uso de mediciones. Las medidas a veces son difíciles de comprender en lo absoluto, aunque en los cambios relativos en las medidas a través de la evolución del sistema puede ser muy informativo porque proveen una base de comparación.

Así el observador puede analizar como las características del producto como el tamaño, la complejidad, el acoplamiento y la cohesión incrementan o disminuyen en el tiempo. También pude monitorearse el cambio relativo de varios aspectos de un producto o pueden proveer los límites de las medidas para apuntar a problemas potenciales y anomalías.

Desventajas

Debido a la interacción con los usuarios finales, cuando sea necesaria la retroalimentación hacia el grupo de desarrollo, utilizar este modelo de desarrollo puede llevar a avances ser extremadamente lento.

Por la misma razón no es una aplicación ideal para desarrollos en los que de antemano se sabe que serán grandes en el consumo de recursos y largos en el tiempo.

Al requerir constantemente la ayuda de los usuario finales, se agrega un costo extra a la compañía, pues mientras estos usuarios evalúan el software dejan de ser directamente productivos para la compañía.

3. CARACTERÍSTICAS DISTINTIVAS DE LAS PRINCIPALES METODOLOGÍAS DE DESARROLLO UTILIZADAS EN LA UNIÓN EUROPEA

3.1. METRICA V3

La metodología MÉTRICA Versión 3 ofrece a las Organizaciones un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del software dentro del marco que permite alcanzar los siguientes objetivos:

- Proporcionar o definir Sistemas de Información que ayuden a conseguir los fines de la Organización mediante la definición de un marco estratégico para el desarrollo de los mismos.

- Dotar a la Organización de productos software que satisfagan las necesidades de los usuarios dando una mayor importancia al análisis de requisitos.

- Mejorar la productividad de los departamentos de Sistemas y Tecnologías de la Información y las Comunicaciones, permitiendo una mayor capacidad de adaptación a los cambios y teniendo en cuenta la reutilización en la medida de lo posible.

- Facilitar la comunicación y entendimiento entre los distintos participantes en la producción de software a lo largo del ciclo de vida del proyecto, teniendo en cuenta su papel y responsabilidad, así como las necesidades de todos y cada uno de ellos.

- Facilitar la operación, mantenimiento y uso de los productos software obtenidos.

La versión 3 de MÉTRICA contempla el desarrollo de Sistemas de Información para las distintas tecnologías que actualmente están conviviendo y los aspectos de gestión que aseguran que un Proyecto cumple sus objetivos en términos de calidad, coste y plazos.

Su punto de partida es la versión anterior de MÉTRICA de la cual se han conservado la adaptabilidad, flexibilidad y sencillez, así como la estructura de actividades y tareas, si bien las fases y módulos de MÉTRICA versión 2.1 han dado paso a la división en Procesos, más ade-

cuada a la entrada-transformación-salida que se produce en cada una de las divisiones del ciclo de vida de un proyecto.

Para cada tarea se detallan los participantes que intervienen, los productos de entrada y de salida así como las técnicas y prácticas a emplear para su obtención.

En la elaboración de MÉTRICA Versión 3 se han tenido en cuenta los métodos de desarrollo más extendidos, así como los últimos estándares de ingeniería del software y calidad, además de referencias específicas en cuanto a seguridad y gestión de proyectos.

También se ha tenido en cuenta la experiencia de los usuarios de las versiones anteriores para solventar los problemas o deficiencias detectados.

En una única estructura la metodología MÉTRICA Versión 3 cubre distintos tipos de desarrollo: estructurado y orientado a objetos, facilitando a través de interfaces la realización de los procesos de apoyo u organizativos:

- Gestión de Proyectos.
- Gestión de Configuración.
- Aseguramiento de Calidad.
- Seguridad.

La automatización de las actividades propuestas en la estructura de MÉTRICA Versión 3 es posible ya que sus técnicas están soportadas por una amplia variedad de herramientas de ayuda al desarrollo.

MÉTRICA Versión 3 tiene un enfoque orientado al proceso, ya que la tendencia general en los estándares se encamina en este sentido y por ello, como ya se ha dicho, se ha enmarcado dentro de la norma ISO 12.207, que se centra en la clasificación y definición de los procesos del ciclo de vida del software. Como punto de partida y atendiendo a dicha norma, MÉTRICA Versión 3 cubre el Proceso de Desarrollo y el Proceso de Mantenimiento de Sistemas de Información.

La metodología descompone cada uno de los procesos en actividades, y éstas a su vez en tareas. Para cada tarea se describe su contenido haciendo referencia a sus principales acciones, productos, técnicas, prácticas y participantes.

El orden asignado a las actividades no debe interpretarse como secuencia en su realización, ya que éstas pueden realizare en orden diferente a su numeración o bien en paralelo.

Sin embargo, no se dará por acabado un proceso hasta no haber finalizado todas las actividades del mismo determinadas al inicio del proyecto.

Así los procesos de la estructura principal de MÉTRICA Versión 3 son los siguientes:

- Planificación de sistemas de información.
- Desarrollo de sistemas de información.
- Mantenimiento de sistemas de información.

Proceso de Planificación de Sistemas de Información

Este proceso al no estar dentro del ámbito de la norma ISO 12.207 de Procesos del Ciclo de Vida de Software, se ha determinado a partir del estudio de los últimos avances en este campo, la alta competitividad y el cambio a que están sometidas las organizaciones. El entorno de alta competitividad y cambio en el que actualmente se encuentran las organizaciones, hace cada vez más crítico el requerimiento de disponer de los sistemas y las tecnologías de la información con flexibilidad para adaptarse a las nuevas exigencias, con la velocidad que demanda dicho entorno.

La existencia de tecnología de reciente aparición, permite disponer de sistemas que apoyan la toma de decisiones a partir de grandes volúmenes de información procedentes de los sistemas de gestión e integrados en una plataforma corporativa. MÉTRICA Versión 3 ayuda en la planificación de sistemas de información facilitando una visión general necesaria para posibilitar dicha integración y un modelo de información global de la organización.

Proceso de Desarrollo de Sistemas de Información

Surge para facilitar la comprensión y dada su amplitud y complejidad se ha subdividido en cinco procesos:

- Estudio de viabilidad del sistema.
- Análisis del sistema de información.
- Diseño del sistema de información.
- Construcción del sistema de información.
- Implantación y aceptación del sistema.

La necesidad de acortar el ciclo de desarrollo de los sistemas de información ha orientado a muchas organizaciones a la elección de productos software del mercado cuya adaptación a sus requerimientos suponía un esfuerzo bastante inferior al de un desarrollo a medida,

por no hablar de los costes de mantenimiento. Esta decisión, que es estratégica en muchas ocasiones para una organización, debe tomarse con las debidas precauciones, y es una realidad que está cambiando el escenario del desarrollo del software.

Otra consecuencia de lo anterior es la práctica, cada vez más habitual en las organizaciones, de la contratación de servicios externos en relación con los sistemas y tecnologías de la información y las comunicaciones, llevando a la necesidad de una buena gestión y control de dichos servicios externos y del riesgo implícito en todo ello, para que sus resultados supongan un beneficio para la organización. MÉTRICA Versión 3 facilita la toma de decisión y la realización de todas las tareas que comprende el desarrollo de un sistema de información.

Proceso de Mantenimiento de Sistemas de Información

Este proceso comprende actividades y tareas de modificación o retirada de todos los componentes de un sistema de información (hardware, software, software de base, operaciones manuales, redes, etc.). Este marco de actuación no es el objetivo de MÉTRICA Versión 3, ya que esta metodología está dirigida principalmente al proceso de desarrollo del software.

Por lo tanto, MÉTRICA Versión 3 refleja los aspectos del Mantenimiento, correctivo y evolutivo, que tienen relación con el Proceso de Desarrollo.

3.2. SEIS SIGMA

Seis Sigma (SS) tuvo sus inicios como una manera de medir y controlar la ocurrencia de defectos en un proceso de productos en masa (elaboración de productos de manufactura).

Seis Sigma incluye un conjunto de herramientas estadísticas, las cuales se han estado perfeccionado durante la última década en una metodología incluyente de aspectos no tan solo estadísticos sino también de estrategias administrativas, filosofías de operación y desempeño.

La metodología Seis Sigma (SS) y Diseño por Seis Sigma (DpSS) han sido aplicadas exitosamente en manufactura, servicios, cadenas de provisión y procesos transaccionales con significativos ahorros y ganancias en términos de eficiencia en dichas industrias.

Sin embargo, su aplicación es incipiente, y más en nuestro país en sectores tales como Tecnologías de Información (TI) y Desarrollo de Software (DSw), a pesar de la necesidad obvia de mejoras radicales en dichas industrias con respecto a calidad y desempeño.

Es relevante mencionar que empresas como GE, Sun Microsystems, Dell, Microsoft, SAP, Cisco, SEI, Boeing, EDS, Oracle, etc., así como eventos internacionales como Six Sigma for IT & Software Development, European Lean Six SigmaEuropean Lean indican una tendencia e interés de aplicar Seis Sigma, Diseño por Seis Sigma, Lean Seis Sigma en Ingenieria de Software y Tecnologías de Información.

El enfoque principal es dar prioridad al cliente. Las mejoras Seis Sigma se evalúan por el incremento en los niveles de satisfacción y creación de valor para el cliente, siempre basado en la realización de productos software acorde a sus requerimientos.

El proceso Six Sigma se inicia estableciendo cuáles son las características críticas de la calidad de software que se deben medir, pasando luego a la recolección de los datos para su posterior análisis. De tal forma, los problemas pueden ser definidos, analizados y resueltos de una forma más efectiva y permanente, atacando las causas raíces o fundamentales que los originan, y no sus síntomas.

3.3. CMMI

El departamento de defensa de los estados unidos tenía muchos problemas con el software que encargaba desarrollar a otras empresas, los presupuestos se disparaban, las fechas alargaban más y más.

Como esta situación les parecía intolerable convocó un comité de expertos para que solucionase estos problemas, dicho comité concluyó “Tienen que crear un instituto de la ingeniería del software, dedicado exclusivamente a los problemas del software, y a ayudar al Departamento de Defensa”.

Convocaron un concurso público en el que dijeron: “Cualquiera que quiera enviar una solicitud tiene que explicar como van a resolver varios de los problemas del software”, se presentaron diversos estamentos y la Universidad Carnegie Mellon ganó el concurso, creando el SEI.

El SEI (Software Engineering Institute) es el instituto que creó y mantiene el modelo de calidad CMM-CMMI.

El CMM-CMMI es un modelo de calidad del software que clasifica las empresas en niveles de madurez. Estos niveles sirven para conocer la madurez de los procesos que se realizan para producir software.

Los niveles CMM-CMMI son 5:

Inicial o Nivel 1 CMM-CMMI. Este es el nivel en donde están todas las empresas que no tienen procesos. Los presupuestos se disparan, no es posible entregar el proyecto en fechas. No hay control sobre el estado del proyecto, **el desarrollo del proyecto es completamente opaco**, no sabes lo que pasa en él.

Repetible o Nivel 2 CMM-CMMI. Quiere decir que el éxito de los resultados obtenidos se pueden repetir. La principal diferencia entre este nivel y el anterior es que **el proyecto es gestionado y controlado durante el desarrollo** del mismo. El desarrollo no es opaco y se puede saber el estado del proyecto en todo momento.

Los procesos que hay que implantar para alcanzar este nivel son:

- Gestión de requisitos.
- Planificación de proyectos.
- Seguimiento y control de proyectos.
- Gestión de proveedores.
- Aseguramiento de la calidad.
- Gestión de la configuración.

Definido o Nivel 3 CMM-CMMI. Alcanzar este nivel significa que la **forma de desarrollar proyectos de desarrollo software (gestión e ingeniería) esta definida**, por definida quiere decir que esta establecida, documentada y que existen métricas (obtención de datos objetivos) para la consecución de objetivos concretos.

Los procesos que hay que implantar para alcanzar este nivel son:

- Desarrollo de requisitos.
- Solución Técnica.
- Integración del producto.
- Verificación.
- Validación.
- Desarrollo y mejora de los procesos de la organización.
- Definición de los procesos de la organización.
- Planificación de la formación.
- Gestión de riesgos.
- Análisis y resolución de toma de decisiones.

La mayoría de las empresas que llegan al nivel 3 paran aquí, ya que es un nivel que proporciona muchos beneficios y no ven la necesidad de ir más allá porque tienen cubiertas la mayoría de sus necesidades.

Cuantitativamente Gestionado o Nivel 4 CMM-CMMI. Los proyectos usan objetivos medibles para alcanzar las necesidades de los clientes y la organización. Se usan métricas para gestionar la organización.

Los procesos que hay que implantar para alcanzar este nivel son:

- Gestión cuantitativa de proyectos.
- Mejora de los procesos de la organización.

Optimizado o Nivel 5 CMM-CMMI. Los procesos de los proyectos y de la organización están orientados a la mejora de las actividades. Mejoras incrementales e innovadoras de los procesos que mediante métricas son identificadas, evaluadas y puestas en práctica.

Los procesos que hay que implantar para alcanzar este nivel son:

- Innovación organizacional.
- Análisis y resolución de las causas.

RESUMEN

El desarrollo de sistemas de software complejos no es una actividad trivial, que pueda abordarse sin un análisis previo. El considerar que un proyecto de desarrollo de software podía abordarse como cualquier otro, ha llevado a una serie de problemas que limitan nuestra capacidad de aprovechar los recursos que el hardware pone a nuestra disposición.

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

Los **métodos** indican cómo construir técnicamente el software, y abarcan una amplia serie de tareas que incluyen la planificación y estimación de proyectos, el análisis de requisitos, el diseño de estructuras de datos, programas y procedimientos, la codificación, las pruebas y el mantenimiento. Los métodos introducen frecuentemente una notación específica para la tarea en cuestión y una serie de criterios de calidad.

Las **herramientas** proporcionan un soporte automático o semiautomático para utilizar los métodos. Existen herramientas automatizadas para cada una de las fases vistas anteriormente, y sistemas que integran las herramientas de cada fase de forma que sirven para todo el proceso de desarrollo. Estas herramientas se denominan CASE (Computer Assisted Software Engineering).

Los **procedimientos** definen la secuencia en que se aplican los métodos, los documentos que se requieren, los controles que permiten asegurar la calidad y las directrices que permiten a los gestores evaluar los progresos.

Existen diferentes ciclos de vida en el desarrollo de SW. Hay un enfoque mas tradicional y algo mas reciente, llamado metodología de desarrollo ágil.

En este último, se encuentra la programación extrema(XP) y también se suele incluir RUP.

Por último mencionar las metodologías mas utilizadas a nivel europeo:

– Metrica V3, principalmente por todas las empresas que trabajan en algún proyecto con la Administración española.
– CMMI, relacionada con ITIL la cual lleva varios años con un auge importante a nivel mundial.
– Seis Sigma, metodología útil en muchos ámbitos de la industria, además del desarrollo del software.