nominados componentes) que sean reutilizables, por un lado, y, por el otro, desarrollar *software* y reutilizar sus fragmentos (que seguramente estarán mejor probados que si se hicieran de nuevo, lo cual además mejoraría la calidad del *software* producido).

Una de las vías mediante las cuales se pretende conseguir una cierta reutilización en el desarrollo orientado a objetos es especialmente con componentes; otras son los patrones de diseño (reutilización, si no de fragmentos de *software*, por lo menos de ideas o "recetas" para hacerlos) y los marcos o *frameworks*, que son estructuras formadas por sistemas de *software* a los cuales se pueden acoplar otros sistemas de *software*, sustituibles, para hacer funciones concretas.

2. El ciclo de vida del *software*

La producción de *software* es algo más que la programación; hay etapas que la preceden y otras que la siguen.

El ciclo de vida del *software* está constituido por el conjunto de todas estas etapas. Los métodos y técnicas de la ingeniería del *software* se inscriben dentro del marco delimitado por el ciclo de vida del *software*, y, más concretamente, por las diferentes etapas que se distinguen.

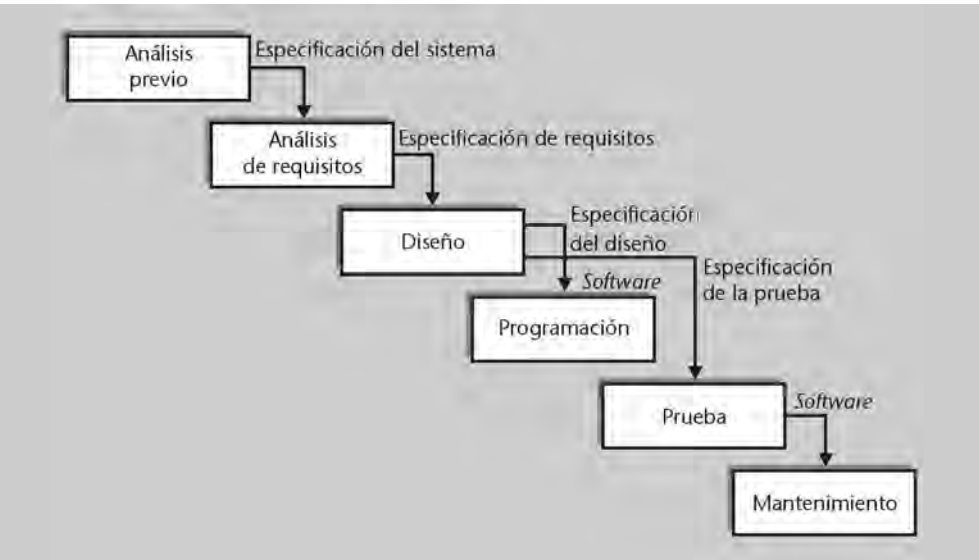
La misma existencia de distintos modelos del ciclo de vida del *software* hace comprender que no hay ninguno que sea ideal o que no tenga grandes limitaciones.

Sin embargo, es indispensable que todo proyecto se desarrolle dentro del marco de un ciclo de vida claramente definido, si se quiere tener una mínima garantía de cumplimiento de los plazos, y respetar los límites de los recursos asignados. Además, la garantía de calidad y las certificaciones* de calidad también presuponen que el proceso de producción de *software* se desarrolle según un ciclo de vida con etapas bien definidas.

*. Un ejemplo de certificación de calidad sería la ISO.

2.1. El ciclo de vida clásico

La figura siguiente nos muestra las etapas previstas en una cierta versión del ciclo de vida clásico.



A veces, el ciclo de vida clásico también se denomina ciclo de vida en cascada, lo cual quiere decir que en cada etapa se obtienen unos documentos (en inglés, *deliverables*) que son las bases de partida de la etapa siguiente –que, por tanto, no puede comenzar antes de que haya terminado la anterior– y nunca se regresa a etapas pasadas.

2.1.1. Etapas

La primera etapa se denomina análisis previo y también análisis de sistemas o ingeniería de sistemas. En esta etapa se definen los grandes rasgos del sistema de *software* que tendrá que dar soporte informático a unas actividades determinadas de unos ciertos usuarios dentro del marco más general de la actividad de la empresa u organización.

Además, este sistema tendrá que funcionar en un entorno de *hardware* y red determinado, que será necesario indicar, y quizá también tendrá que intercambiar información con otro *software* o compartir una base de datos. Estos hechos constituyen otros aspectos del entorno del futuro *software* de los cuales se tendrá que dejar constancia.

Hay que tener en cuenta los recursos necesarios para el desarrollo del *software* y los condicionamientos temporales, especialmente los plazos impuestos desde fuera del proyecto, que a menudo están determinados por los hechos que han causado las necesidades de información que tiene que satisfacer dicho *software*, y también restricciones eventuales y condiciones adicionales que sea necesario respetar; y, en función de todo esto, se evalúa la viabilidad técnica, económica y legal del proyecto de desarrollo de dicho *software*.

El documento que resulta de esta etapa se denomina Especificación del sistema, y sirve de base para tomar la decisión definitiva sobre la continuación del proyecto.

La segunda etapa es el análisis de requisitos o simplemente análisis. Su objetivo es definir con detalle las necesidades de información que tendrá que resolver el *software*, sin tener en cuenta, por el momento, los medios técnicos con los que se tendrá que llevar a término el desarrollo del *software*.

Como el lenguaje de programación, el gestor de bases de datos, los componentes que se pueden reutilizar, etc.

En esta etapa detallamos los requisitos de la etapa anterior; ahora sólo pensamos en el *software* que es necesario desarrollar y sus interfaces con el entorno.

La figura responsable del análisis –el analista, que puede ser un informático o un usuario– debe tener o adquirir conocimientos generales sobre el dominio de la aplicación y obtener información de los usuarios y de otras fuentes que le permita hacerse una idea precisa de las funciones, y de los requisitos en general, del futuro *software*. Con esta información se redacta el documento que llamaremos Especificación de requisitos, que tiene una doble función: especificar qué debe hacer el *software*, con la suficiente precisión para que se pueda desarrollar, y servir de base para un contrato, explícito o no, entre el equipo de desarrollo del *software* y sus futuros usuarios.

El diseño es la etapa siguiente. Si el análisis especifica el problema o "qué tiene que hacer el *software*", el diseño especifica una solución a este problema o "cómo el *software* tiene que hacer su función".

Del *software*, hay que diseñar varios aspectos diferenciados: su arquitectura general, las estructuras de datos (base de datos, etc.), la especificación de cada programa y las interfaces con el usuario, y se tiene que llevar a cabo de manera que, a partir de todo esto, se pueda codificar el *software*, de una manera parecida a la construcción de un edificio o de una máquina a partir de unos planos.

El documento resultante es la *Especificación del diseño*. La etapa de diseño es el mejor momento para elaborar la *Especificación de la prueba*, que describe con qué datos se tiene que probar cada programa o grupo de programas y cuáles son los resultados esperados en cada caso.

La programación o codificación, que es la cuarta etapa, consiste en traducir el diseño a código procesable por el ordenador.

Es en esta etapa donde se le da forma real al *software*, es en realidad cuando se elabora.

El entregable que se genera en esta etapa es el programa propiamente, con todas sus funcionalidades y componentes.

La prueba es la última etapa del desarrollo del *software* y la penúltima del modelo de ciclo de vida del *software* que hemos considerado.

La etapa de prueba consiste en probar el *software* desde distintos puntos de vista de una manera planificada y, naturalmente, localizar y corregir dentro del *software* y su documentación los errores que se detecten.

La prueba se lleva a término en las dos fases siguientes:

1) En la primera se hacen pruebas, primero para cada uno de los programas por separado y, después, por grupos de programas directamente relacionados, y se aplica la especificación de la prueba que hemos mencionado con anterioridad.

2) En la segunda fase se comprueba que el conjunto de programas dé los resultados que se esperan y que lo haga con el rendimiento deseado.

El primer equipo de desarrollo hace la última fase de la prueba y, si los resultados son satisfactorios, se entrega el *software* al cliente, el cual puede hacer una prueba parecida por su cuenta y con sus datos, con la finalidad de decidir si acepta el *software*. Con la aceptación por parte del cliente se da por terminado el desarrollo.

La última etapa del ciclo de vida es el mantenimiento o, si se prefiere, explotación, del *software*, ya que siempre que se utilice el *software* habrá que mantenerlo, es decir, hacer cambios –pequeños o grandes– para corregir errores, mejorar

las funciones o la eficiencia, o adaptarlo a un nuevo *hardware* o a cambios en las necesidades de información.

Puesto que un *software* puede estar en explotación diez años o más, a menudo el coste total del mantenimiento durante la vida del *software* es de dos a cinco veces mayor que el coste de desarrollo.

2.1.2. El caso de lenguajes de cuarta generación

Los lenguajes –o más bien entornos de programación– de cuarta generación son de muy alto nivel (en el sentido de que a menudo una sola de sus instrucciones equivale a muchas instrucciones del lenguaje ensamblador) y en gran parte son no procedimentales y están integrados en un gestor de bases de datos relacionales. Incluyen herramientas de dibujo de pantallas, generación de listados y en ocasiones salidas gráficas y hoja de cálculo. Algunos pueden generar código en un lenguaje de tercera generación.

Para aplicaciones sencillas, se puede pasar directamente de los requisitos a la codificación, pero en proyectos complejos es necesario llevar a cabo una etapa de diseño, aunque simplificada.

2.2. Los ciclos de vida iterativos e incrementales

El ciclo de vida en cascada ha sido muy criticado y se han propuesto algunos modelos alternativos.

2.2.1. Inconvenientes del modelo de ciclo de vida en cascada

El inconveniente del modelo de ciclo de vida en cascada es que no es realista.

Como se ha visto, el modelo de ciclo de vida en cascada comporta que las sucesivas etapas del desarrollo se hacen de manera lineal, de forma que una fase no comienza mientras no se haya acabado la anterior, y no se vuelve nunca atrás. También queda implícito en el modelo que, cuando se acaba una fase, se sabe al menos aproximadamente qué porcentaje del proyecto queda por hacer, ya que

si el análisis se ha completado y su resultado es cien por cien fijo, se puede saber con cierta precisión la duración del diseño e, incluso, de la programación.

Ahora bien, en la realidad es posible que la especificación del sistema sea fiable en lo que respecta a las funciones, ya que no se espera que se describan punto por punto. Sin embargo, precisamente por esto último, el coste y la duración del proyecto se han calculado sobre una base muy poco sólida y tienen un gran margen de error.

No obstante, el problema más grave se presenta en el análisis de requisitos, por el hecho de que éstos casi siempre son incompletos al principio o cambian antes de que se haya acabado de construir el *software*, y a menudo suceden ambas cosas a la vez. Y si la especificación de requisitos es incompleta e insegura, es obvio que el diseño y la programación tendrán problemas y, sobre todo, retrasos y aumentos de coste importantes para el trabajo no previsto que se deberá hacer y también para el que será necesario rehacer.

Existen dos razones por las cuales es prácticamente imposible elaborar unos requisitos completos y estables en el primer intento:

a) En primer lugar, es difícil encontrar un conjunto de futuros usuarios que conozcan lo suficiente el entorno en el que se debe utilizar el *software*, que hayan reflexionado lo suficiente sobre lo que quieren conseguir y que, además, se pongan de acuerdo.

b) En segundo lugar, porque el trabajo de consolidación de las peticiones de estos usuarios nunca será perfecto.

En cualquier caso, tenemos que contar con el hecho de que, una vez terminada oficialmente la etapa de análisis y comenzada la de diseño, todavía surgirán requisitos nuevos y cambios en los ya existentes.

¿Qué se puede hacer, entonces? Parece que la opción más razonable sea estudiar a fondo una pequeña parte de los requisitos que tenga una cierta autonomía, y diseñarla, programarla y probarla, y una vez que el cliente la haya dado por buena, hacer lo mismo con otra parte, y otra. Si partimos de un *software* ya construido en parte, se puede esperar que la idea que nos hacemos de los requisitos restantes pueda ser cada vez más precisa y que también obtengamos una estimación cada vez más segura del coste y de la duración del proyecto completo; esto es lo que denominamos ciclo de vida iterativo e incremental (iterativo por-

que se repite dentro de un mismo proyecto e incremental porque procede por partes). Y se tendrá que considerar normal que, a veces, cuando se construya una parte, se vea que es necesario modificar una hecha con anterioridad.

La necesidad de estimar el coste y los plazos

Los inconvenientes del modelo del ciclo de vida clásico mencionados en este subapartado no quieren decir que no pueda haber plazo o límite de coste para un proyecto de desarrollo de *software*; simplemente, se debe reconocer que no es realista creer que se pueda fijar de forma exacta la funcionalidad, el coste y la duración del proyecto, todo a la vez. Si el *software* tiene que funcionar en una fecha determinada y no se puede aumentar el gasto en personal, será necesario estar dispuesto a aceptar que el *software* no realice todas las funciones deseadas; si la funcionalidad y la fecha de entrega del programa son innegociables, se tendrá que aumentar el número de programadores o analistas. Y esto no supone ninguna renuncia en relación con los resultados que se alcanzaban hasta ahora, porque en la práctica muy pocos eran los proyectos en los que no se producían desviaciones en cuanto a la funcionalidad, presupuesto o plazo, sino en varias de estas cosas al mismo tiempo.

Por tanto, el modelo de ciclo de vida en cascada puede ser válido si se aplica de manera que cada etapa, del análisis de requisitos a la prueba, no prevea todo el conjunto del *software*, sino sólo una parte cada vez; entonces tendríamos un ciclo de vida iterativo e incremental basado en el ciclo de vida en cascada.

2.2.2. El ciclo de vida con prototipos

Para ayudar a concretar los requisitos, se puede recurrir a construir un prototipo del *software*.

Un prototipo es un *software* provisional, construido con herramientas y técnicas que dan prioridad a la rapidez y a la facilidad de modificación antes que a la eficiencia en el funcionamiento, que sólo tiene que servir para que los usuarios puedan ver cómo sería el contenido o la apariencia de los resultados de algunas de las funciones del futuro *software*.

Un prototipo sirve para que los usuarios puedan confirmar que lo que se les muestra, efectivamente, es lo que necesitan o bien lo puedan pedir por comparación, y entonces se prepara una nueva versión del prototipo teniendo en cuen-

ta las indicaciones de los usuarios y se les enseña otra vez. En el momento en que el prototipo ha permitido concretar y confirmar los requisitos, se puede comenzar un desarrollo según el ciclo de vida en cascada, en este caso, no obstante, partiendo de una base mucho más sólida.

Características del ciclo de vida con prototipos

El ciclo de vida con prototipos no se puede considerar plenamente un ciclo de vida iterativo e incremental, ya que sólo el prototipo se elabora de manera iterativa, y no necesariamente incremental. Sin embargo, es un modelo de ciclo de vida que puede ser adecuado en algunos casos, en especial cuando basta con prototipar un número reducido de funciones para que las otras sean bastante parecidas a éstas de forma que las conclusiones a las que se llegue con el prototipo también les sean aplicables.

2.2.3. La programación exploratoria

La programación exploratoria consiste en elaborar una primera versión del *software*, o de una parte de éste, enseñarla a los usuarios para que la critiquen y, a continuación, hacerle los cambios que éstos sugieran, proceso que se repetirá tantas veces como sea necesario.

La diferencia principal con respecto a los prototipos es que aquí el *software* es real desde el principio.

Características de la programación exploratoria

La programación exploratoria se puede considerar un ciclo de vida iterativo, pero no incremental, ya que el *software* está completo desde la primera versión. Como consecuencia de las numerosas modificaciones que sufre, la calidad del *software* desarrollado de esta manera y de su documentación tiende a ser deficiente, como la de un *software* que haya experimentado un mantenimiento largo e intenso.

2.2.4. El ciclo de vida del *Rational Unified Process*

La empresa Rational Software ha propuesto este ciclo de vida como marco para el desarrollo de *software* que utiliza sus herramientas. Es claramente un ciclo de vida iterativo e incremental.

Se distinguen estas cuatro etapas (denominadas *fases*):

1) Inicio

Se establece la justificación económica del *software* y se delimita el alcance del proyecto.

2) Elaboración

Se estudia el dominio del problema, o simplemente dominio (parte de la actividad de la empresa dentro de la cual se utilizará el *software*) y se tienen en cuenta muchas de las necesidades de información y eventuales requisitos no funcionales y restricciones, se establece la arquitectura general del *software* y se realiza una planificación del proyecto.

3) Construcción

Se desarrolla todo el producto de forma iterativa e incremental, se tienen en cuenta todas las necesidades de información que debe satisfacer y se desarrolla la arquitectura obtenida en la fase anterior.

4) Transición

Comprende la entrega del producto al cliente y el comienzo de su utilización; aunque es posible que sea necesario hacer retoques en el *software* y añadir nuevas funciones como consecuencia de errores detectados o de requisitos que se habían pasado por alto hasta el momento.

En cada una de estas fases se llevan a cabo (en diferentes proporciones) los siguientes componentes de proceso:

- recogida de requisitos (*requirement capture*),
- análisis y diseño,
- realización (*implementation*),
- prueba (*test*).

Cada unidad en la que se ejecutan pocos o muchos de los componentes de proceso es una iteración, y se aplica a un nuevo fragmento de *software*. Todas las fases tienen iteraciones.