Tema 2. Ingeniería de software. Ciclos de vida

2.1.- Modelos del ciclo de vida del software

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

Cada una de estas etapas lleva asociada una serie de tareas que deben realizarse, y una serie de documentos (en sentido amplio: software) que serán la salida de cada una de estas fases y servirán de entrada en la fase siguiente. Existen diversos modelos de ciclo de vida, es decir, diversas formas de ver el proceso de desarrollo de software. En este apartado veremos algunos de los principales modelos de ciclo de vida. 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.

2.1.1.- Paradigma del ciclo de vida en cascada

El paradigma del ciclo de vida en cascada 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.

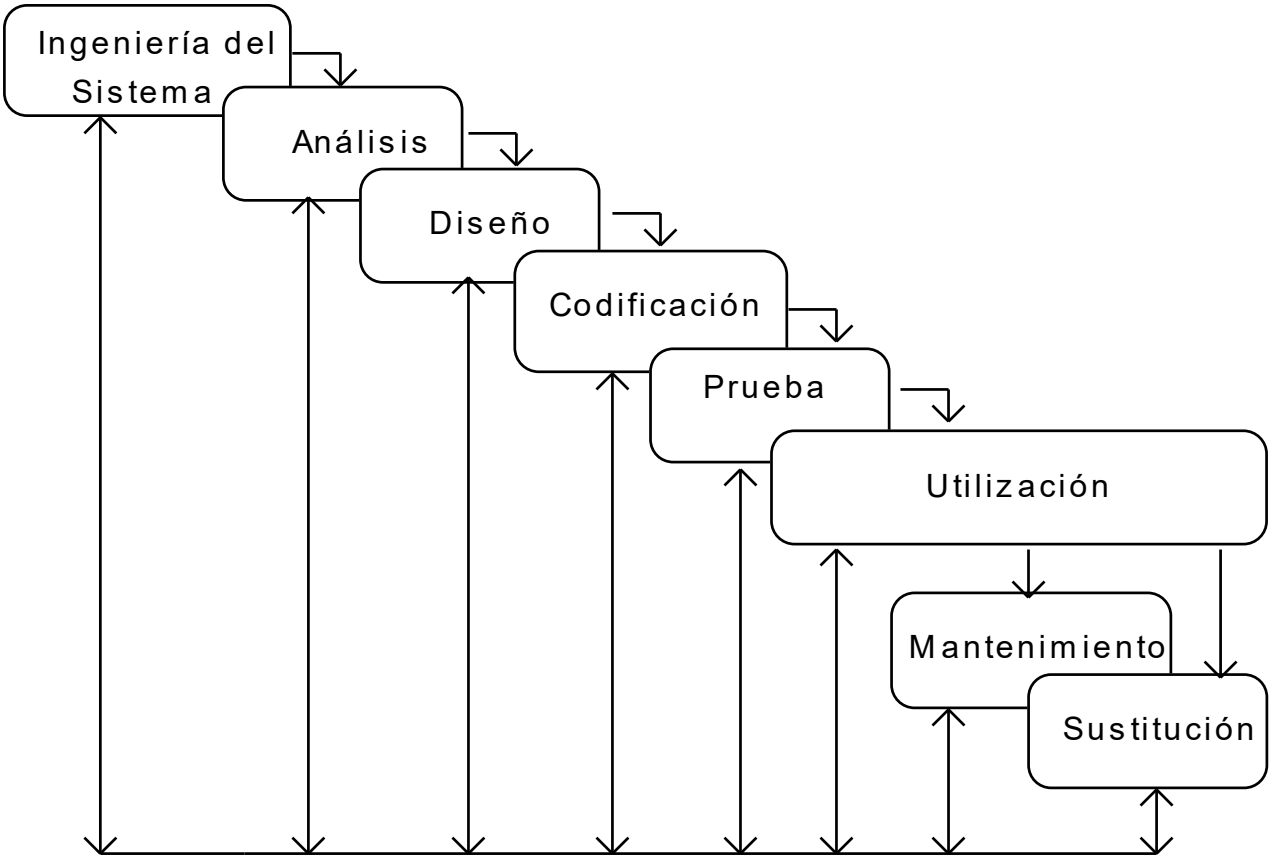

Figura 2.1. Ciclo de vida en cascada.

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

2.1.1.1.- Ingeniería y análisis del sistema.

El software es siempre parte de un sistema mayor, por lo que siempre va a interrelacionarse con otros elementos, ya sea hardware, máquinas o personas. Por esto, el primer paso del ciclo de vida de un proyecto consiste en un análisis de las características y el comportamiento del sistema del cual el software va a formar parte. 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 asignaremos un subconjunto de estos requisitos al software. En el caso de un sistema ya existente (supongamos, por ejemplo, que queremos informatizar una empresa) deberemos 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.

2.1.1.2.- Análisis de requisitos del software.

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

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

2.1.1.3.- 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).

2.1.1.4.- Codificación.

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

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 éste 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.

2.1.1.5.- Prueba.

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

2.1.1.6.- Utilización.

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

2.1.1.7.- Mantenimiento.

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

Que, durante la utilización, el cliente detecte errores en el software: los errores latentes.

Que se produzcan cambios en alguno de los componentes del sistema informático: por ejemplo cambios en la máquina, en el sistema operativo o en los periféricos.

Que el cliente requiera modificaciones funcionales (normalmente ampliaciones) no contempladas en el proyecto.

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

El modelo en cascada, a pesar de ser lineal, contiene flujos que permiten la vuelta atrás. Así, desde el mantenimiento se vuelve al análisis, el diseño o la codificación, y también desde cualquier fase se puede volver a la anterior si se detectan fallos. Estas vueltas atrás no son controladas, ni quedan explícitas en el modelo, y este es uno de los problemas que presenta este paradigma

2.1.1.8.- Sustitución.

La vida del software no es ilimitada y cualquier aplicación, por buena que sea, acaba por ser sustituida por otra más amplia, rápida, bonita o 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 implementadores, que tienen que corregir los errores que aparecen. Es necesario hacer un trasvase de la información que maneja el sistema viejo a la estructura y el formato requeridos por el nuevo.

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

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

El ciclo de vida en cascada es el paradigma más antiguo, más conocido y todavía ampliamente usado en la IS. No obstante, ha sufrido diversas críticas, debido a los problemas que se plantean al intentar aplicarlo a determinadas situaciones. Entre estos problemas están:

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.

2.1.2.- Paradigma de la 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. En general, cualquier aplicación que presente mucha interacción con el usuario, o que necesite algoritmos que puedan construirse de manera evolutiva, yendo de lo más 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'.

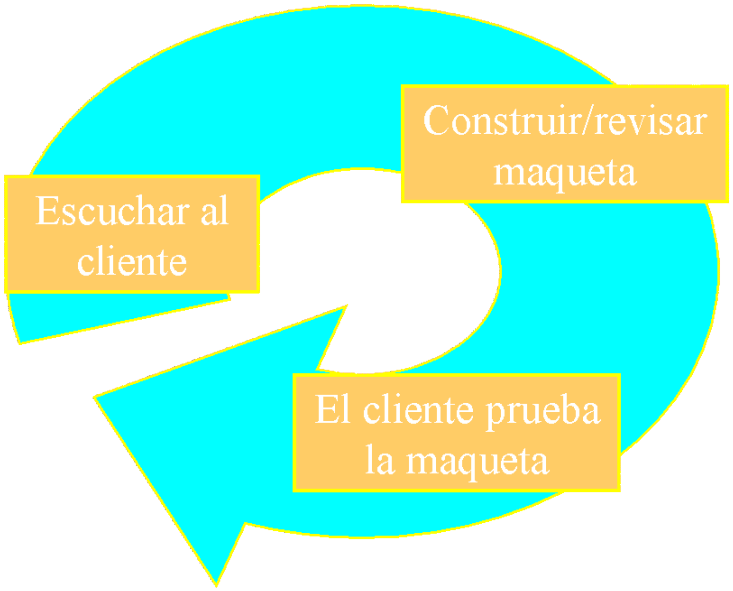

Figura 2.2. Filosofía de desarrollo del paradigma de prototipos

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, como análisis de alternativas, para decidir, antes de empezar la fase de diseño, cuál es el modelo más adecuado, de entre la gama disponible, para el soporte hardware o cómo deben hacerse los accesos a la base de datos para obtener buenas respuestas en tiempo cuando la aplicación esté ya en funcionamiento.

En otros casos, el prototipo servirá para modelar y poder mostrar al cliente cómo va a realizarse la E/S de datos en la aplicación, de forma que éste pueda hacerse una idea de como va a ser el sistema final, pudiendo entonces detectar deficiencias o errores en la especificación, aunque el modelo no sea más que una cáscara vacía.

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

- un prototipo, en papel o ejecutable en ordenador, que describa la interacción hombre-máquina.
- un prototipo que implemente algún(os) subconjunto(s) de la función requerida, y que sirva para evaluar el rendimiento de un algoritmo o las necesidades de capacidad de almacenamiento y velocidad de cálculo del sistema final.
- un programa que realice en todo o en parte la función deseada pero que tenga

características (rendimiento, consideración de casos particulares, etc.) que deban ser mejoradas durante el desarrollo del proyecto.

La secuencia de tareas del paradigma de construcción de prototipos puede verse en la figura 2.5.

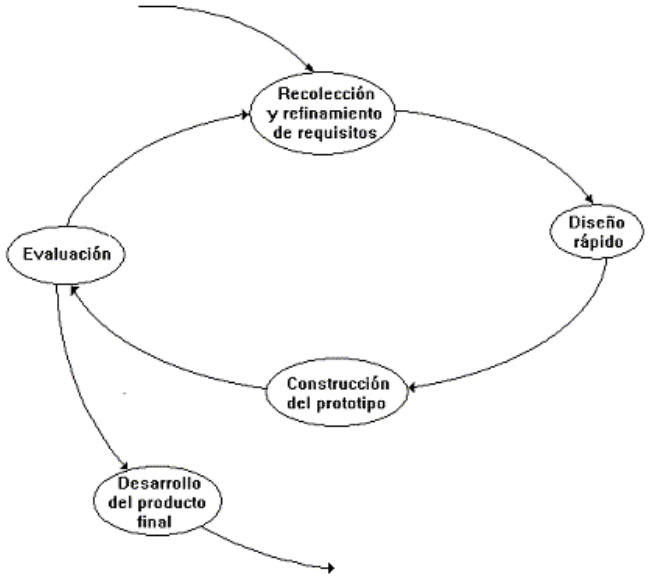

Figura 2.3. Construcción de prototipos

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 procedimentales 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. Sea cual sea la opción elegida, el objetivo es que permita la codificación rápida y facilite el desarrollo incremental de las especificaciones conforme estas se van completando en cada ciclo. En este punto es irrelevante la calidad del software generado siempre que no impida poner a prueba las especificaciones.

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, que habíamos realizado cuidadosamente, pensando que estábamos desarrollando el producto final. Al tener que repetir estas fases, sí que estaremos desechando una gran cantidad de trabajo, normalmente muy superior al esfuerzo de construir un prototipo basándose en un diseño rápido, en la reutilización de trozos de software preexistentes y en herramientas de generación de código para informes y manejo de ventanas.

Precisamente, 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.- Ciclo de vida incremental

El modelo incremental combina elementos del modelo lineal en cascada con la filosofía iterativa de construcción de prototipos. En el modelo incremental (ver figura) se va creando el sistema software añadiendo componentes funcionales al sistema (llamados incrementos). En cada paso sucesivo, se actualiza el sistema con nuevas funcionalidades o requisitos, es decir, cada

versión o refinamiento parte de una versión previa y le añade nuevas funcionalidades. El sistema software, igual que el diseño por prototipos, ya no se ve como un producto con una fecha de entrega sino como una integración resultado de sucesivos refinamientos. Sin embargo, a diferencia del modelo por prototipos el modelo incremental se centra en obtener un producto operativo en cada iteración, de forma que los primeros incrementos son versiones incompletas del producto final pero proporciona al usuario una plataforma de evaluación con la que empezar a trabajar.

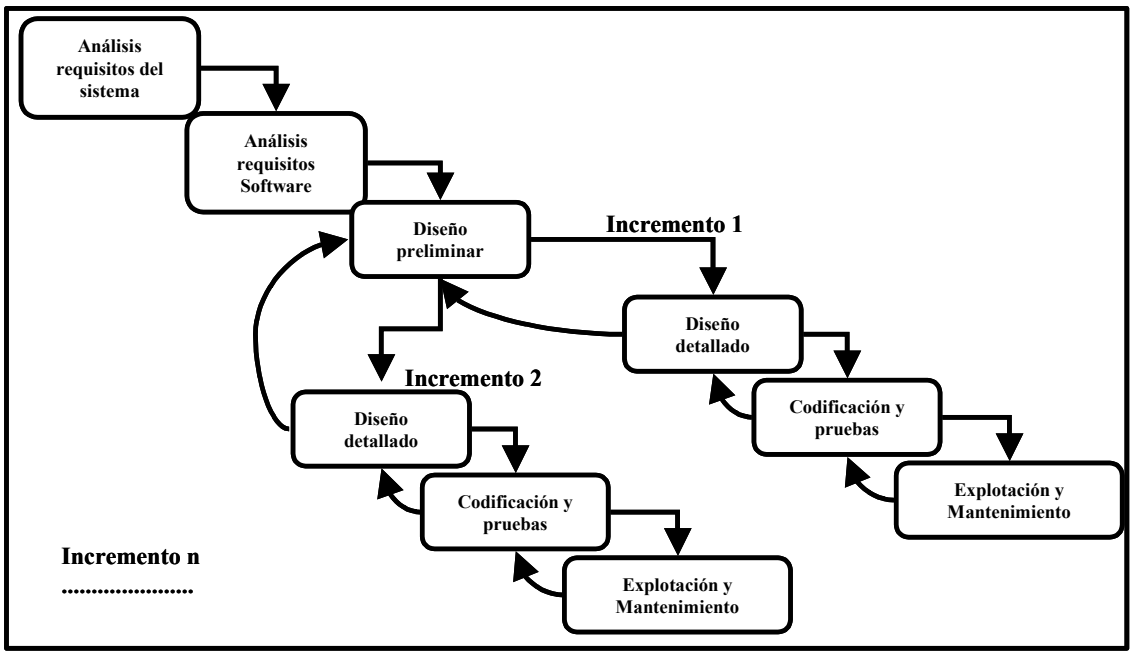

Figura 2.4. El modelo incremental.

El modelo incremental, al igual que el de prototipos, se ajusta a entornos de fuerte incertidumbre en la que se debe comenzar el sistema sin poder realizar una especificación exhaustiva o bien el personal no está disponible para una implementación completa en una fecha en la que debemos tener una versión operativa.

2.1.4.- Uso de técnicas de cuarta generación.

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

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

Acceso a bases de datos utilizando lenguajes de consulta de alto nivel (derivados normalmente de SQL). Con ello no es necesario conocer la estructura de los ficheros o tablas ni de sus índices.

Generación de código. A partir de una especificación de los requisitos se genera automáticamente toda la aplicación.

Generación de pantallas. Permiten diseñar la pantalla dibujándola directamente, incluyendo además el control del cursor y la gestión de errores de los datos de entrada.

Gestión de entornos gráficos.

Generación de informes. (de forma similar a las pantallas).

Esta generación automática permite reducir la duración de las fases del ciclo de vida clásico, especialmente la fase de codificación, quedando el ciclo de vida según se indica en la figura 3.5.

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.).

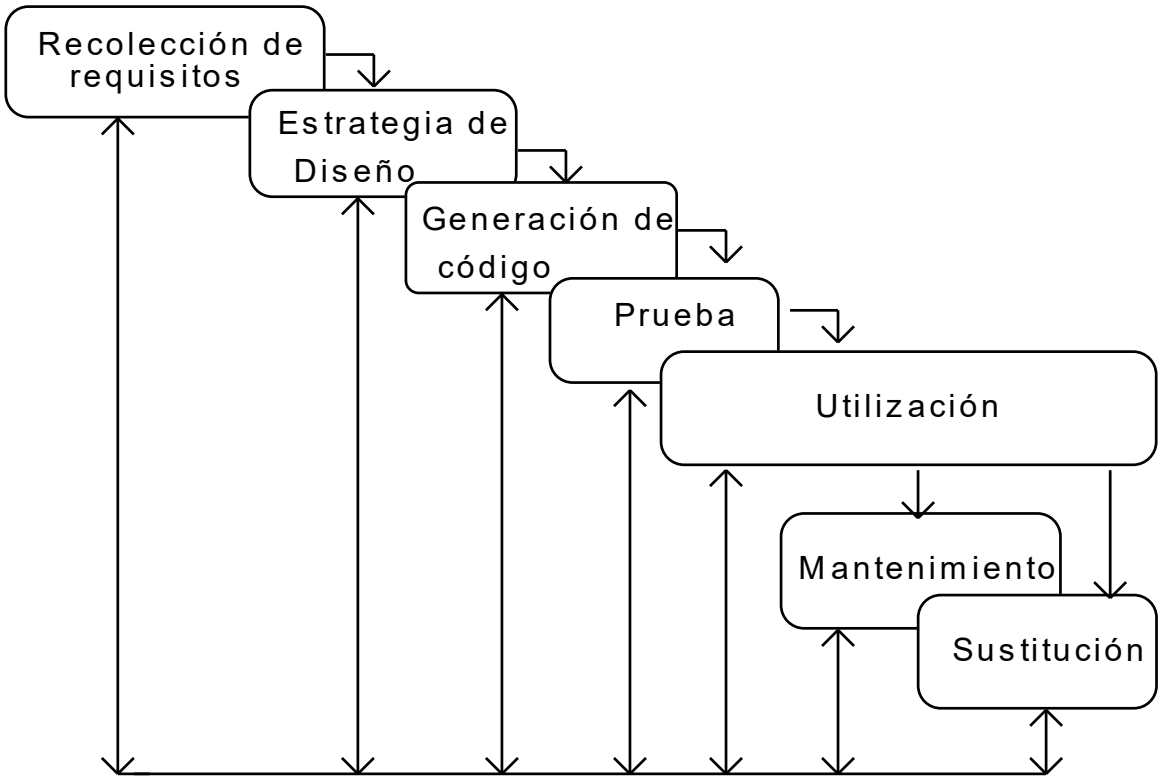

Figura 2.5. Ciclo de vida usando técnicas de cuarta generación.

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 (aunque nunca hay que fiarse) que el código generado es correcto y acorde con la especificación, y que no contiene los típicos errores de la codificación manual. Pero en cualquier caso es necesaria la fase de pruebas, en primer lugar para comprobar la eficiencia del código generado (la generación automática de los accesos a bases puede producir código muy 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.: Informix, Natural).

No obstante, entre las críticas más habituales están:

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

El código fuente que producen es ineficiente.(el ejemplo de antes de SQL). Al estar generado automáticamente no pueden hacer uso de los *trucos* habituales para aumentar el rendimiento, que se basan en el buen conocimiento de cada caso particular. 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.

2.1.5.- El modelo en espiral.

Boehm (1988) propuso un marco del proceso de software dirigido por el riesgo (el modelo en espiral) que es muestra en la figura 3.6. 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.

La principal característica del modelo en espiral 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.

2.1.5.1.- Descripción del modelo

El proceso de software se representa como una espiral, y no como una secuencia de actividades con cierta vinculación de una actividad a otra. Cada ciclo en la espiral representa una fase del proceso de software. El ciclo más interno puede relacionarse con la factibilidad del sistema, el siguiente ciclo con la definición de requisitos, el ciclo que sigue con el diseño del sistema, etc. El modelo en espiral combina el evitar el cambio con la tolerancia al cambio. Lo anterior supone que los cambios son resultado de riesgos del proyecto e incluye actividades de gestión de riesgos explicitas para reducir tales riesgos. El modelo en espiral define cuatro sectores, y representa cada uno de ellos en un cuadrante:

- ***Establecimiento de objetivos***: Se definen objetivos específicos para la fase actual del proyecto y las restricciones con las que estos deben alcanzarse. Finalmente se identifican y proponen distintas alternativas que permitirían alcanzar los objetivos con sus restricciones.
- ***Análisis de riesgo***: El cuadrante siguiente se centra en los riesgos. En este punto el modelo propone la identificación de los riesgos de las diferentes alternativas y la formulación de una estrategia, efectiva en coste, para resolver dichos riesgos.
- ***Desarrollo y validación:*** Si en el cuadrante anterior hemos reducido el riesgo a valores razonables en este se realizan las actividades de ingeniería que construyen el producto. En el modelo original de Boehm las actividades de este cuadrante seguirían el ciclo de vida en cascada pero el modelo es fácilmente ampliable para seleccionar cualquier otro de los vistos hasta ahora.
- ***Revisión y Planificación:*** Consiste en la revisión de todos los productos desarrollados en el ciclo y en ella intervienen todas las personas que tienen relación con el producto. Con esta revisión se realizan los planes para la ejecución del siguiente ciclo. La revisión de los principales objetivos sirve para asegurar que todas las partes involucradas están de acuerdo respecto al método de trabajo para la fase siguiente. El proceso finaliza con la toma de decisión de si debe finalizar o continuarse el proyecto y con los compromisos adquiridos para la siguiente fase por todos sus interesados.

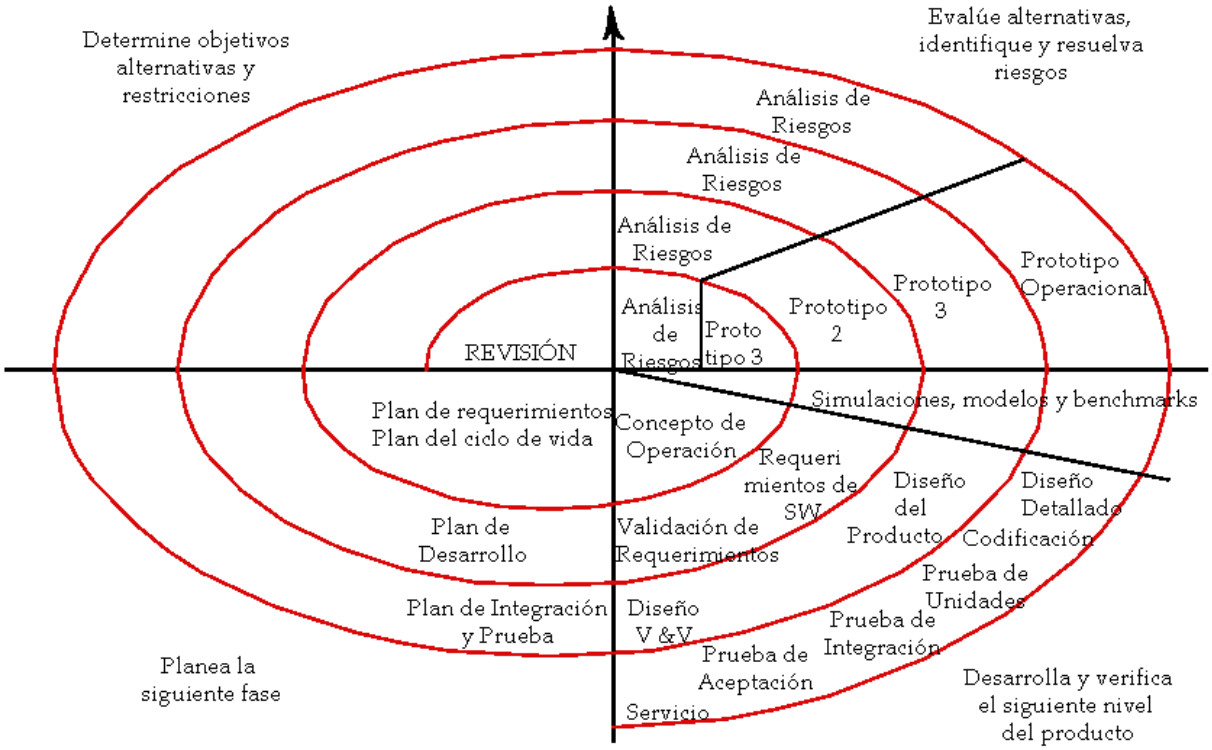

Figura 2.6. El modelo en espiral.

En su artículo original Boehm explica su modelo a través de un ejemplo en el que curiosamente se añade un ciclo inicial, que no está incluido en la figura 2.6, en el que, en el cuadrante de ingeniería, se realizaría un *análisis de viabilidad* del objetivo propuesto, que sería anterior al *concepto de operación* definido en el primer ciclo de la figura.

El ejemplo comienza proponiéndose el **objetivo** de aumentar la productividad de la empresa en el desarrollo de software respetando como **restricciones** que el coste para alcanzarlo fuera razonable y que no se cambiase la cultura de empresa. Para ello se identifican varias **alternativas:** en el área de gestión, mejorar la organización de los proyectos, su control o su planificación; en el área de personal, añadir formación o incentivos; mejorar las instalaciones o en el área tecnológica, cambiar herramientas o máquinas o incluir el reuso. En este punto es importante señalar que la mayoría de las alternativas no implican ningún desarrollo software.

A continuación, el modelo propone la **identificación de riesgos** que en este caso supone que las ganancias en productividad no fueran significativas o no fuera posible conservar, con las mejoras propuestas, la cultura de empresa. Para **minimizar estos riesgos** se propone la revisión del estado del arte o encuestas internas.

Como se comentó anteriormente, como actividad del tercer sector se realiza el ***análisis de la viabilidad*** de las alternativas que permite eliminar alguna de ellas por ser irrealizable y que la combinación de algunas de ellas puede producir los resultados deseados por lo que es necesario un estudio en profundidad para determinar la combinación más eficaz.

En la parte final del modelo, en el último cuadrante de la espiral, se desarrolla un plan para realizar encuestas y análisis más extensos, desarrollar el concepto de operación y la justificación económica. La fase se cierra con el compromiso alcanzado de financiar la siguiente fase.

Una vez realizado el primer ciclo se volvería a empezar. En el segundo ciclo, en el que se

realizaría el concepto de operación de lo que se pretende construir, se definen objetivos y restricciones mucho más específicos y medibles. De esta forma podría ponerse como objetivos que la productividad se pudiera doblar al cabo de 5 años con la restricción de coste de implementación de 10.000€ por persona y manteniendo la cultura de empresa.

Las principales diferencias entre el modelo en espiral y los métodos más tradicionales son:

- Existe un reconocimiento explícito de las diferentes alternativas para alcanzar los objetivos de un proyecto
- La identificación de riesgos asociados con cada una de las alternativas y las diferentes maneras de resolverlos son el centro del modelo. Con los métodos tradicionales, es habitual dejar las partes más difíciles para el final y empezar con las más fáciles y de menor riesgo, obteniendo así la ilusión de un gran avance.
- La división de los proyectos en ciclos, cada uno con un acuerdo al final de cada ciclo, implica que existe un acuerdo para los cambios que hay que realizar o para terminar el proyecto, en función de lo que se ha aprendido desde el inicio del proyecto.
- El modelo se adapta a cualquier tipo de actividad, incluidas algunas que no existen en otros métodos (por ejemplo, consulta de asesores expertos o investigadores ajenos) que son muy útiles para la consecución de los objetivos de un proyecto.

El modelo en espiral puede aplicarse en la mayoría de las ocasiones. Sin embargo, en algunos casos hay que resolver ciertas dificultades:

- Trabajo con software contratado. El modelo en espiral trabaja bien en los desarrollos internos, pero necesita un ajuste posterior para adaptarlo a la subcontratación de software. En el desarrollo interno existe una gran flexibilidad y libertad para ajustarse a los acuerdos etapa por etapa, para aplazar acuerdos de opciones específicas, para establecer miniespirales para resolver caminos críticos, para ajustar niveles de esfuerzo, o para acomodar prácticas como prototipado, desarrollo evolutivo, o uso de métodos de diseño ajustado al coste. En el desarrollo de software bajo contrato no existe esta flexibilidad y libertad, por lo que es necesario mucho tiempo para definir los contratos, ya que los entregables no estarán previamente definidos de forma clara.
- Necesidad de expertos en evaluación de riesgos para identificar y manejar las fuentes de riesgos de un proyecto. Normalmente, un equipo sin experiencia puede producir una especificación con una gran elaboración de los elementos de bajo riesgo bien comprendidos, y una pequeña y pobre elaboración de los elementos de alto riesgo. A no ser que se realice una inspección por expertos, en este tipo de proyecto se tendrá la ilusión de progresar durante un período, y, sin embargo, se encuentra dirigida directamente hacia el desastre. Otro aspecto a tener en cuenta es que una especificación dirigida por riesgo es también dependiente del personal. Por ejemplo, un diseño producido por un experto puede ser implantado por inexpertos. Sin embargo, lo contrario es muy difícil llevarlo a cabo.

Hoy en día la gestión de riesgo ha salido del entorno de aplicación del ciclo de vida en espiral para transformarse en un proceso de peso propio en el desarrollo de un proyecto software. Por este motivo, introducimos en este punto un apartado cuya brevedad puede hacer engañosa su importancia que fácilmente lo haría merecedor de un tema. Aunque este apartado está extraído de Sommerville, 2005 una aproximación mucho más detallada del problema la encontramos en CMU/SEI-93-TR-6.

2.1.5.2.- Análisis de riesgos [Sommerville, 2005]

Una de las actividades de la planificación de proyectos, tal y como se ha visto en la descripción de las normas, es el análisis de riesgos. De forma simple, se puede concebir un riesgo como una probabilidad de que una circunstancia adversa ocurra. Los riesgos son una amenaza para el proyecto, para el software que se está desarrollando y para la organización.

La gestión de riesgos es particularmente importante para los proyectos de software debido a las incertidumbres inherentes con las que se enfrentan muchos proyectos. Estas incertidumbres son el resultado de los requerimientos ambiguamente definidos, las dificultades en la estimación de tiempos y los recursos para el desarrollo del software, la dependencia en las habilidades individuales, y los cambios en los requerimientos debidos a los cambios en las necesidades del cliente. Es preciso anticiparse a los riesgos: comprender el impacto de éstos en el proyecto, en el producto y en el negocio, y considerar los pasos para evitarlos. En el caso de que ocurran, se deben crear planes de contingencia para que sea posible aplicar acciones de recuperación. El análisis de riesgos consiste en cuatro actividades principales:

- *Identificar los riesgos*. Identificar los posibles *riesgos del proyecto* (afectan a la calendarización y los recursos. etc.), *riesgos del producto* (problemas de diseño, codificación, mantenimiento), *riesgos del negocio* (riesgos de mercado: que se adelante la competencia o que el producto no se venda bien).
- *Análisis 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.
- *Planificación 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).

El proceso de gestión de riesgos, como otros de planificación de proyectos, es un proceso iterativo que se aplica a lo largo de todo el proyecto. Una vez que se genera un conjunto de planes iniciales, se supervisa la situación. En cuanto surja más información acerca de los riesgos, éstos deben analizarse nuevamente y se deben establecer nuevas prioridades. La prevención de riesgos y los planes de contingencia se deben modificar tan pronto como surja nueva información de los riesgos.

Los resultados del proceso de gestión de riesgos se deben documentar en un plan de gestión de riesgos. Éste debe incluir un estudio de los riesgos a los que se enfrenta el proyecto, un análisis de éstos y los planes requeridos para su gestión. Si es necesario, puede incluir algunos resultados de la gestión de riesgos; por ejemplo, planes específicos de contingencia que se activan si aparecen dichos riesgos.

Identificación de riesgos

Ésta es la primera etapa de la gestión de riesgos. Comprende el descubrimiento de los posibles riesgos del proyecto. En principio, no hay que valorarlos o darles prioridad en esta etapa aunque, en la práctica, por lo general no se consideran los riesgos con consecuencias menores o con baja probabilidad.

Esta identificación se puede llevar a cabo a través de un proceso de grupo utilizando un enfoque de tormenta de ideas o simplemente puede basarse en la experiencia. Para ayudar al proceso, se utiliza una lista de posibles tipos de riesgos. Hay al menos seis tipos de riesgos que pueden aparecer:

1. Riesgos de tecnología. Se derivan de las tecnologías de software o de hardware utilizadas en el sistema que se está desarrollando.
2. Riesgos de personal. Riesgos asociados con las personas del equipo de desarrollo.
3. Riesgos organizacionales. Se derivan del entorno organizacional donde el software se está desarrollando.
4. Riesgos de herramientas. Se derivan de herramientas CASE y de otro software de apoyo utilizado para desarrollar el sistema.
5. Riesgos de requerimientos. Se derivan de los cambios de los requerimientos del cliente y el proceso de gestionar dicho cambio.
6. Riesgos de estimación. Se derivan de los estimados administrativos de las características del sistema y los recursos requeridos para construir dicho sistema.

Análisis de riesgos

Durante este proceso, se considera por separado cada riesgo identificado y se decide acerca de la probabilidad y la seriedad del mismo. No existe una forma fácil de hacer esto -recae en la opinión y experiencia del gestor del proyecto-. No se hace una valoración con números precisos sino en intervalos, por ejemplo:

- La probabilidad del riesgo se puede valorar como muy bajo, bajo, moderado, alto o muy alto.
- Los efectos del riesgo pueden ser valorados como catastrófico, serio, tolerable o insignificante.

En ambos casos las etiquetas subjetivas así definidas como muy bajo, bajo,… para la probabilidad o Catastrófico, serio,.. para el impacto deberían definirse de acuerdo a una escala con criterios objetivos como la probabilidad muy alta se establece para aquellos riesgos que con casi total seguridad se darán en los siguientes dos meses del proyecto o un riesgo tolerable será aquel cuyo impacto no excede de los colchones temporales o económicos disponibles para ese punto del proyecto.

El resultado de este proceso de análisis se debe colocar en una tabla, la cual debe estar ordenada según la seriedad del riesgo. Para hacer esta valoración se necesita información detallada del proyecto, el proceso, el equipo de desarrollo y la organización.

Por supuesto, tanto la probabilidad como la valoración de los efectos de un riesgo cambian conforme se disponga de mayor información acerca del riesgo y los planes de gestión del mismo se imp1ementen. Por lo tanto, esta tabla se debe actualizar durante cada iteración del proceso de riesgos.

Una vez que los riesgos se hayan analizado y clasificado, se debe discernir cuáles son los más importantes que se deben considerar durante el proyecto. Este discernimiento debe depender de una combinación de la probabilidad de aparición del riesgo y de los efectos del mismo. En

general, siempre se deben tener en cuenta todos los riesgos catastróficos, así como todos los riesgos serios que tienen más que una moderada probabilidad de ocurrir.

Boehm (Boehm, 1988) recomienda identificar y supervisar los «10 riesgos más altos», pero este número parece demasiado arbitrario. El número exacto de riesgos a supervisar debe depender del proyecto. Pueden ser cinco o 15. No obstante, el número apropiado debe ser manejable. Un número muy grande de riesgos requiere obtener mucha información.

Planificación de riesgos

El proceso de planificación de riesgos considera cada uno de los riesgos clave que han sido identificados, así como las estrategias para gestionados. Otra vez, no existe un proceso sencillo que nos permita establecer los planes de gestión de riesgos. Depende del juicio y de la experiencia del gestor del proyecto. Las estrategias seguidas pueden dividirse en tres categorías.

1. *Estrategias de prevención*. Siguiendo estas estrategias, la probabilidad de que el riesgo aparezca se reduce.
2. *Estrategias de minimización*. Siguiendo estas estrategias se reducirá el impacto del riesgo.
3. *Planes de contingencia*. Seguir estas estrategias es estar preparado para lo peor y tener una estrategia para cada caso.

Puede verse aquí una analogía con las estrategias utilizadas en sistemas críticos para asegurar fiabilidad, protección y seguridad. Básicamente, es mejor usar una estrategia para evitar el riesgo. Si esto no es posible, utilizar una para reducir los efectos serios de los riesgos. Finalmente, tener estrategias para reducir el impacto del riesgo en el proyecto y en el producto.

Supervisión de riesgos

La supervisión de riesgos normalmente valora cada uno de los riesgos identificados para decidir si éste es más o menos probable y si han cambiado sus efectos. Por supuesto, esto no se puede observar de forma directa, por lo que se tienen que buscar otros factores para dar indicios de la probabilidad del riesgo y sus efectos. Obviamente, estos factores dependen de los tipos de riesgo.

La supervisión de riesgos debe ser un proceso continuo y, en cada revisión del progreso de gestión, cada uno de los riesgos clave debe ser considerado y analizado por separado.

2.2.- Desarrollo ágil.

[Pressman 6ª, T-4] [Sommerville, T-17] En todo lo visto hasta ahora debe destacarse el esfuerzo por documentar los proyectos software. No se trata, como hemos visto, únicamente de comentar el software sino de un esfuerzo totalmente distinto al de codificar. Se trata por un lado de describir el software en un alto nivel de abstracción que permita analizar su estructura o comportamiento sin entrar en el detalle de la codificación y aún comprobar que hará aquello que se le pide antes de escribir la primera línea de código. Por otro, se definen también esfuerzos que no están orientados a la construcción de la aplicación sino a la búsqueda de sus posibles fallos y, por ende, a su registro y al seguimiento de las correcciones y modificaciones que estos suponen. Aún más se han discutido esfuerzos orientados a garantizar la calidad del producto en la rigurosidad de sus procesos de construcción. Por si todo ello fuera poco ninguno de estos documentos tiende a permanecer inmutable en el tiempo lo que hace preciso el seguimiento de sus modificaciones.

Todo este esfuerzo se orienta a sistematizar la construcción del software para que sea lo más repetible y predecible posible y por consiguiente que podamos predecir que esfuerzo (coste) y

calidad (errores) podemos esperar de nuestro software. A cambio, el esfuerzo realizado reduce la velocidad de desarrollo hasta puntos que pueden resultar inaceptables. Aparece entonces el concepto de peso de una metodología asociada al volumen de la notación, el grado de formalismo, el tamaño de los modelos y la dificultad para su mantenimiento conforme se producen cambios.

Por contraposición el desarrollo ágil o ligero trata de aliviar el peso de sus metodologías y defiende la renuncia expresa a utilizar modelos perfectos, se busca únicamente que sean lo suficientemente buenos. Esta afirmación incluso se condiciona al grupo de desarrollo de forma que lo que es suficiente para un grupo puede ser insuficiente o incluir aspectos innecesarios para otro. En definitiva tratan de centrar los esfuerzos en presentar un incremento del software ejecutable restando importancia a los productos del trabajo intermedio lo que no siempre es bueno.

En el manifiesto para el desarrollo ágil del software firmado por Kent Beck y otros 16 notables desarrolladores, escritores y consultores en el 2001 se establecía:

Hemos descubierto mejores formas de desarrollar software al construirlo por nuestra cuenta y ayudar a otros a hacerlo. Por medio de este trabajo hemos llegado a valorar:

- *A los individuos y sus intenciones sobre los procesos y sus herramientas*
- *Al software en funcionamiento sobre la documentación extensa*
- *A la colaboración del cliente sobre la negociación del contrato*
- *A la respuesta al cambio sobre el seguimiento de un plan*

Esto es, aunque los términos de la derecha tienen valor, nosotros valoramos más los aspectos de la izquierda

En definitiva se construye una corriente contrapuesta a los modelos prescriptivos de procesos vistos con anterioridad. Esto, sin embargo, no debe entenderse como la eliminación de un modelo para imponer otro. Se trata simplemente de destacar que existen limitaciones y defectos que deben ser corregidos en los modelos planteados y que estos no siempre se adecuan a todos los desarrollos. Dos de los grandes problemas que se aprecian en los modelos anteriores tienen que ver con su limitada capacidad para adaptarse a los cambios y con la disciplina que exigen en el trabajo de las personas.

Un hecho destacado por las ingenierías ligeras es la necesidad de adaptarse al cambio. Este se produce por varios factores de entre los que destaca la fragilidad de los requisitos y las prioridades del cliente. Tales cambios no dependen únicamente de limitaciones en el cliente para definir con precisión sus necesidades sino en la variabilidad del entorno en el que este se mueve que puede cambiar de forma impredecible sus necesidades. Pero la variabilidad del proceso no está sólo en el cliente, con frecuencia resulta impredecible cuanto diseño es preciso antes de poder construir de forma que, en general, análisis, diseño y construcción presentan variaciones imprevisibles desde el punto de vista de la planificación.

La variabilidad descrita tendería a transformar los proyectos en una adaptación continua sin progreso. Para enfrentarse a este problema las metodologías ligeras proponen una aproximación incremental en la que se valora la entrega frecuente. Cada incremento puede desarrollarse en un par de semanas o un par de meses primando siempre la escala de tiempo más corta. Los incrementos son valorados por el cliente que entra de forma efectiva en el proceso del software evitando el nocivo “nosotros y vosotros” y ofrece la necesaria realimentación al equipo de desarrollo proporcionándole la información que este necesita sobre sus nuevos requisitos o prioridades.

El otro gran problema de las metodologías pesadas es el personal de desarrollo. Estas metodologías se enfrentan a las limitaciones de las personas que intervienen en los procesos exigiendo disciplina para garantizar que los procesos se realizan de acuerdo al modelo de trabajo

que se ha planteado. Según señala Alistair Cockburn esto plantea una debilidad inherente a estas metodologías pues exigen del personal un funcionamiento más propio de robots que de personas. Los errores del personal en la consistencia de su trabajo se transmiten a la calidad de los resultados de la metodología. Por ejemplo, un documento incorrectamente actualizado llevará a errores futuros cuando sea preciso su uso. Obviamente este problema se puede enfrentar con la disciplina del personal, pero la debilidad propia de su humanidad se traduce en la fragilidad de los modelos.

Las metodologías ligeras se enfrentan a este problema mediante un enfoque de tolerancia aún admitiendo que probablemente este sea menos productivo. Se plantea de esta forma que el modelo se adapte al equipo y no a la inversa, tal y como ya indicábamos en el comienzo del apartado. Como consecuencia el éxito del desarrollo va a estar muy condicionado a la calidad del equipo por lo que deben buscarse un gran número de rasgos:

Competencia. En el contexto de un desarrollo ágil (al igual que en la ingeniería del software convencional), la "competencia" abarca un talento innato, habilidades específicas relacionadas con el software, y un conocimiento general del proceso que el equipo haya elegido aplicar. La habilidad y el conocimiento del proceso pueden y deben enseñarse a toda la gente que funge como miembro de un equipo ágil.

Enfoque común. Aunque los miembros del equipo ágil desempeñen tareas diferentes y aporten distintas habilidades al proyecto, todos deben enfocarse en una meta: entregar al cliente un incremento de trabajo de software dentro del tiempo establecido. Alcanzar esta meta requiere que el equipo también se centre en adaptaciones continuas (pequeñas y grandes) mediante las cuales el proceso satisfará las necesidades del equipo.

Colaboración. La ingeniería del software (sin considerar el proceso) incluye evaluar, analizar y usar información que se comunica al equipo de software; crear información que ayudará al cliente y a otros a entender el trabajo del equipo; y construir información (software de computadora y bases de datos relevantes) que ofrezca un valor comercial para el cliente. Estas tareas se cumplirán si los miembros del equipo colaboran, entre ellos, con el cliente y con sus gerentes.

Habilidad para la toma de decisiones. Todo buen equipo de software (incluidos los equipos ágiles) debe permitirse la libertad de controlar su propio destino. Esto implica que al equipo se le dé autonomía, es decir, autoridad para tomar decisiones en cuanto a cuestiones técnicas y del proyecto.

Capacidad de resolución de problemas confusos. Los gestores de software deben reconocer que el equipo ágil enfrentará ambigüedades y sufrirá golpes de manera continua debido al cambio. En algunos casos, el equipo debe aceptar que el problema que está resolviendo hoy tal vez no sea el problema que debe resolverse mañana. Sin embargo, las lecciones aprendidas en cualquier actividad para la resolución de problemas (incluidas aquellas que sirven para solucionar el problema erróneo) pueden beneficiar al equipo en fases posteriores del proyecto.

Confianza y respeto mutuo. El equipo ágil se debe convertir en lo que De Marco y Lister [DEM98] llaman un equipo "cuajado" (véase el tema 21 Pressman). Un equipo cuajado muestra la confianza y el respeto necesarios para que "se unan con tanta fuerza, que el todo sea mayor que la suma de las partes" [DEM98].

Organización propia. En el contexto del desarrollo ágil, la organización propia implica tres factores: 1) el equipo ágil se organiza a sí mismo para el trabajo que debe hacerse; 2) el equipo organiza el proceso que mejor se ajusta a su ambiente local; 3) el equipo organiza el programa de

trabajo con el que se alcance de mejor manera la entrega del incremento del software. La organización propia tiene varios beneficios técnicos, pero lo más importante es que mejora la colaboración y eleva moral del equipo. En esencia, el equipo sirve como su propia gestoría. Ken Schwaber [SCH02] puntualiza estos aspectos cuando escribe: "El equipo selecciona la cantidad de trabajo que cree que es capaz de hacer dentro de la iteración, y el equipo se compromete con el trabajo. Nada desalienta más a un equipo que alguien más se comprometa por él. Nada motiva más a un equipo que aceptar la responsabilidad de cumplir los compromisos que él mismo hizo".

El objetivo que se persigue en la ingeniería ligera o ágil es como su nombre indica la agilidad. Se trata pues de viajar con poco equipaje para que se nos facilite la reacción a los cambios frecuentes de forma que aunque ligereza y agilidad no hagan referencia al mismo aspecto si están directamente relacionados. La alianza ágil define 12 principios para el que quiere alcanzar la agilidad

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software valioso.
2. Bienvenidos los requisitos cambiantes, incluso en fases tardías del desarrollo. La estructura de los procesos ágiles cambia para la ventaja competitiva del cliente.
3. Entregar con frecuencia software en funcionamiento, desde un par de semanas hasta un par de meses, con una preferencia por la escala de tiempo más corta.
4. La gente de negocios y los desarrolladores deben trabajar juntos a diario a lo largo del proyecto.
5. Construir proyectos alrededor de individuos motivados. Darles el ambiente y el soporte que necesitan, y confiar en ellos para obtener el trabajo realizado.
6. El método más eficiente y efectivo de transmitir información hacia y dentro de un equipo de desarrollo es la conversación cara a cara.
7. El software en funcionamiento es la medida primaria de progreso.
8. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben ser capaces de mantener un paso constante de manera indefinida.
9. La atención continua a la excelencia técnica y al buen diseño mejora la agilidad.
10. La simplicidad -el arte de maximizar la cantidad de trabajo no realizado- es esencial.
11. Las mejores arquitecturas, los mejores requisitos y los mejores diseños emergen de equipos autoorganizados.
12. A intervalos regulares el equipo refleja la forma en que se puede volver más efectivo; entonces su comportamiento se ajusta y adecua en concordancia.

En este punto cabe destacar que en la actualidad existe cierto debate entre las virtudes y defectos de los dos tipos de ingenierías, cada una de las cuales tiene sus partidarios y detractores. Igual que con otros aspectos de la Ingeniería del Software cabe esperar evoluciones y nuevos planteamientos que tomen los aspectos positivos de cada filosofía dejando que el contexto y el problema a resolver sea lo que condicione el modelo que nos acercará más a una u otra. De hecho, igual que sucede con las ingenierías anteriores también en este caso existen varios modelos de proceso.

La programación extrema (PE), el desarrollo adaptativo del software (DAS), métodos de desarrollo de sistemas dinámicos, la melé, Cristal, Desarrollo conducido por Características (DCC), Modelado ágil. Todos estos modelos siguen los principios de la Alianza Ágil y presentan características muy semejantes y tratan de organizar grupos cuajados con objetivos muy concretos que pueden desarrollarse en breves espacios de tiempo para presentar al cliente un incremento con funcionalidades usables a la espera de que éste defina nuevas prioridades. De todos ellos destacamos la programación extrema por ser uno de los que ha alcanzado el mayor índice de popularidad.

2.2.1.- Modelado Ágil

El modelado ágil (MA) es una tecnología basada en la práctica para el modelado efectivo de los sistemas basados en software. Dicho de una forma más simple, el modelado ágil es una colección de valores, principios y prácticas para el modelado de software que puede aplicarse en un proyecto de desarrollo de software de una manera efectiva y ligera. Los modelos ágiles son más efectivos que los tradicionales porque son sólo lo suficientemente buenos, no tienen que ser perfectos [AMB021:

Además de los valores consistentes con el manifiesto ágil, Ambler sugiere valor y humildad. Un equipo ágil debe tener el valor para tomar decisiones que ocasionarán el rechazo y la refabricación de un diseño. Debe tener la humildad de reconocer que quienes manejan la tecnología no tienen todas las respuestas, y que los expertos en negocios y otros participantes de la empresa son dignos de respeto y consideración.

El MA sugiere un amplio conjunto de principios de modelado "esenciales" y "suplementarios" de los que cabe destacar

Modelar con un propósito. Un desarrollador que use el MA debe tener una meta específica en mente (por ejemplo, comunicar información al cliente o ayudarle a entender mejor algún aspecto del software) antes de crear el modelo. Una vez identificada la meta para el modelo, el tipo de notación que se usará y el grado de detalle requerido serán más obvios.

Usar múltiples modelos. Existen muchos modelos y notaciones diferentes con los cuales describir el software. Sólo un pequeño subconjunto es esencial para la mayoría de los proyectos. El MA sugiere que para proporcionar la visión necesaria cada modelo debe presentar un aspecto diferente del sistema, y sólo aquellos modelos que proporcionen un valor para la audiencia a la que están dirigidos deben usarse.

Viajar ligero. La realización de trabajo de la ingeniería del software requiere conservar sólo los modelos que proporcionarán valor a largo plazo y descartar el resto. Cada producto de trabajo que se conserve debe recibir mantenimiento conforme se presentan cambios. Esto representa un trabajo que reduce la velocidad del equipo. Ambler [AMB02] observa que "cada vez que se decide conservar un modelo se intercambia la agilidad por la conveniencia de tener la información disponible para el equipo de una forma abstracta (por ende, existe una posibilidad de mejorar la comunicación dentro del equipo, así como con los propietarios del proyecto)".

El contenido es más importante que la representación. El modelado debe comunicar información a la audiencia a la que está dirigido. Un modelo sintácticamente perfecto que comunique sólo un poco del contenido útil no tiene tanto valor como un modelo con una notación defectuosa que, sin embargo, comunique un contenido valioso para su audiencia.

Conocer los modelos y las herramientas con que se crean. Es necesario entender las fortalezas y debilidades de cada modelo y las herramientas con los que se creó.

Adaptar en forma local. El enfoque del modelado se debe adaptar a las necesidades del equipo ágil.

2.2.2.- Programación extrema

A pesar de que los primeros trabajos sobre las ideas y los métodos asociados con la

programación extrema (PE) se realizaron a finales de la década de 1980, el trabajo fundamental sobre la materia, escrito por Kent Beck [BEC99], se publicó en 1999. Los libros subsiguientes de Jeffries et al. [JEF01] sobre los detalles técnicos de la PE, y el trabajo adicional de Beck y Fowler [BEC01b] sobre la planeación de la PE expusieron los detalles del método.

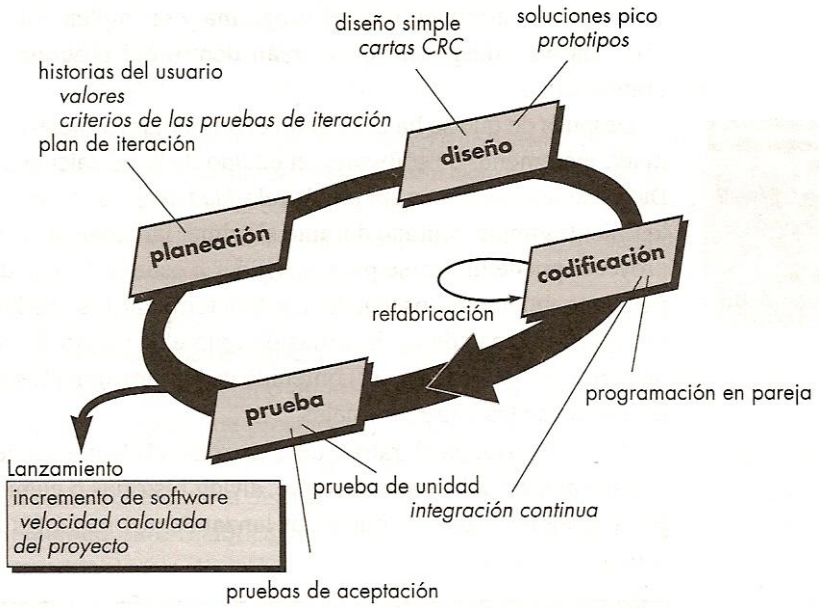

Figura 2.7. Ciclo de la programación extrema

La PE utiliza un enfoque orientado a objetos como su paradigma de desarrollo preferido. La PE abarca un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades del marco de trabajo: planificación, diseño, codificación y pruebas. En la figura se ilustra el proceso de la PE y se observan algunas de las ideas y tareas clave asociadas con cada actividad del marco de trabajo. En los siguientes párrafos se resumen las actividades clave de la PE.

Planificación. La actividad de planificación comienza creando una serie de historias (también llamadas historias del usuario) que describen las características y la funcionalidad requeridas para el software que se construirá. Cada historia (similar a los casos de uso) la escribe el cliente y se coloca en una carta índice. El cliente le asigna un valor (es decir, una prioridad) a la historia basándose en los valores generales del negocio respecto de la característica o la función. Su valor también puede depender de la presencia de otra historia. Los miembros del equipo de la PE evalúan entonces cada historia y le asignan un costo, el cual se mide en semanas de desarrollo. Si la historia requiere más de tres semanas de desarrollo, se le pide al cliente que la divida en historias menores, y se realiza de nuevo la asignación del valor y el costo. Es importante destacar que las historias nuevas pueden escribirse en cualquier momento.

Los clientes y el equipo de PE trabajan juntos para decidir cómo agrupar las historias hacia el próximo lanzamiento (el siguiente incremento de software) para que el equipo de la PE las desarrolle. Una vez establecido el compromiso básico (el acuerdo de las historias que se incluirán, la fecha de entrega y otras cuestiones del proyecto) para un lanzamiento, el equipo de la PE ordena las historias que se desarrollarán de una de las siguientes tres maneras: 1) todas las historias serán implementadas de un modo inmediato (dentro de pocas semanas); 2) las historias con valor más alto se moverán en el programa y se implementarán al principio; o 3) las historias de mayor riesgo se moverán dentro del programa y se implementarán al principio.

Después de que se ha entregado el primer lanzamiento del proyecto (también llamado incremento de software), el equipo de la PE calcula la velocidad del proyecto. Dicho de un modo más simple, la velocidad del proyecto es el número de historias de los clientes implementado durante el primer lanzamiento. Entonces, la velocidad del proyecto puede utilizarse para 1) ayudar a estimar fechas de entrega y el programa para lanzamientos subsecuentes, y 2) determinar si se ha hecho un compromiso excesivo en algunas de las historias de todo el proyecto de desarrollo. Si se presenta un compromiso excesivo, el contenido de los lanzamientos se modifica o se cambian las fechas de las entregas finales.

Conforme avanza el trabajo de desarrollo, el cliente puede agregar historias, cambiar el valor de la historia existente, dividir historias o eliminarlas. Entonces el equipo de la PE considera de nuevo los lanzamientos restantes y modifica sus planes de acuerdo con ello.

Diseño. El diseño de la PE sigue de manera rigurosa el principio MS (mantenerlo simple). Siempre se prefiere un diseño simple respecto de una presentación más compleja. Además, el diseño ofrece una guía de implementación para una historia como está escrita, ni más ni menos. Se desaprueba el diseño de funcionalidades extra (porque el desarrollador supone que se requerirá más tarde).

La PE apoya el uso de tarjetas CRC (colaborador-responsabilidad-clase) como un mecanismo efectivo para pensar en el software en un contexto orientado a objetos. Las tarjetas CRC identifican y organizan las clases orientadas al objeto que son relevantes para el incremento del software actual. Las tarjetas CRC son el único producto de trabajo realizado como parte del proceso de la PE.

Si se encuentra un problema difícil de diseño como parte del diseño de la historia, la PE recomienda la creación inmediata de un prototipo operacional de esa porción del diseño. El prototipo del diseño, llamado la solución pico, se implementa y evalúa. El propósito es reducir el riesgo cuando comience la verdadera implementación y validar las estimaciones originales en la historia que contiene el problema del diseño.

La PE apoya la refabricación, una técnica de construcción que también lo es de diseño. Fowler [FOWOO] describe la refabricación de la siguiente manera:

Refabricación es el proceso de cambiar un sistema de software de tal manera que no altere el comportamiento externo del código y que mejore la estructura interna. Es una manera disciplinada de limpiar el código [y modificar/simplificar el diseño interno], lo que minimiza las oportunidades de introducir errores. En esencia, al refabricar, se mejora el diseño del código después de que se ha escrito.

Debido a que el diseño de la PE virtualmente no utiliza la notación y produce, si acaso, muy pocos productos de trabajo, distintos a las tarjetas de CRC y soluciones pico, el diseño se considera como un artefacto que puede y debe modificarse de manera continua a medida que prosigue la construcción. El propósito de refabricar es controlar estas modificaciones al sugerir pequeños cambios del diseño que "pueden mejorar de manera radical el diseño" [FOWOO]. Sin embargo, debe notarse que el esfuerzo requerido para refabricar puede aumentar en forma drástica a medida que crece el tamaño de la aplicación.

Una noción central en la PE es que el diseño ocurre tanto antes como después del comienzo de la codificación. Refabricar significa que el diseño ocurre de manera continua a medida que se construye el sistema. De hecho, la actividad de construcción misma le proporcionará al equipo de PE una guía sobre cómo mejorar el diseño.

Codificación. La PE recomienda que después de diseñar las historias y realizar el trabajo de diseño preliminar el equipo no debe moverse hacia la codificación, sino que debe desarrollar una serie de pruebas de unidad que ejerciten cada una de las historias que vayan a incluirse en el lanzamiento actual (incremento de software). Una vez creada la prueba de unidad, el desarrollador es más capaz de centrarse en lo que debe implementarse para pasar la prueba de unidad, igual que si se conocen las preguntas de un examen resulta más sencillo saber que estudiar. No se agrega nada extraño (MS). Una vez que el código está completo, la unidad puede probarse de inmediato, y así proporcionar una retroalimentación instantánea a los desarrolladores.

Un concepto clave durante la actividad de codificación (y uno de los aspectos de la PE de los que más se ha hablado) es la programación en pareja. La PE recomienda que dos personas trabajen juntas en una estación de trabajo de computadora para crear el código de una historia. Esto proporciona un mecanismo para la resolución de problemas en tiempo real (dos cabezas piensan mejor que una) y el aseguramiento de la calidad en las mismas condiciones. También alienta que los desarrolladores se mantengan centrados en el problema que se tiene a la mano. En la práctica, cada persona tiene un papel sutilmente diferente. Por ejemplo, una persona puede pensar en los detalles de codificación de una porción particular del diseño, mientras que la otra se asegura de que se sigan los estándares de codificación (una parte requerida de la PE) y que el código que se genera "coincida" con el diseño más amplio de la historia.

Cuando los programadores completan su trabajo el código que desarrollaron se integra con el trabajo de otros. En algunos casos esto lo lleva a cabo diariamente el equipo de integración. En otros casos, la pareja de programadores es la responsable de la integración. Esta estrategia de "integración continua" ayuda a evitar problemas de compatibilidad e interfaz y proporciona un ambiente de "prueba de humo" que ayuda a descubrir los errores desde el principio. Una prueba de humo es una prueba de integración que ejercita todo el sistema de forma que sin ser exhaustiva tiene una alta probabilidad de detectar errores importantes.

Pruebas. Ya se ha hecho notar que la creación de una prueba de unidad antes de comenzar la codificación es un elemento clave para el enfoque de la PE. Las pruebas de unidad que se crean deben implementarse con un marco de trabajo que permita automatizarlas (por lo tanto, pueden ejecutarse de manera fácil y repetida). Esto apoya una estrategia de regresión de prueba cuando el código se modifica (al cual a menudo se le confiere la filosofía de la PE de refabricar).

Cuando las unidades individuales de prueba se organizan en un "conjunto universal de pruebas" [WEL99], las pruebas de integración y validación del sistema puede; realizarse a diario. Esto proporciona al equipo de PE una indicación continua de progreso y también puede encender luces de emergencia previas si las cosas salen mal. Wells [WEL99] establece: "Arreglar problemas pequeños cada pocas horas supone menos tiempo que arreglar problemas enormes justo antes de la fecha límite".

Las pruebas de aceptación de la PE, también llamadas pruebas del cliente, las especifica el cliente y se enfocan en las características generales y la funcionalidad de sistema, elementos visibles y revisables por el cliente. Las pruebas de aceptación Se derivan de las historias del usuario que se han implementado como parte de un lanzamiento de software.