Tema 1. El producto 1

Tema 1. El producto

1.1.- Introducción.

La situación actual en los sistemas informáticos se caracteriza por una rápida evolución de los componentes hardware, que incrementan continuamente sus prestaciones manteniendo o incluso disminuyendo sus precios, junto con una fuerte tendencia a la estandarización e interoperabilidad (ordenadores personales, estaciones de trabajo con sistema operativo UNIX, sistemas distribuidos funcionando sobre plataformas heterogéneas, dispositivos móviles, etc.) y una gran diversidad de marcas y modelos con prestaciones y precios similares. En este escenario, la potencia de los grandes ordenadores de las décadas pasadas está hoy disponible en un mini ordenador e incluso en un ordenador personal. El software es el mecanismo que nos permite utilizar y explotar este potencial.

Esto hace que, a la hora de plantearnos la adquisición de un sistema informático completo, ya sea para gestionar una empresa, para controlar un proceso industrial, o para uso doméstico, el software es lo que marca la diferencia. Entre varios productos de características hardware similares, nos decidiremos por una determinada compañía vendedora basándonos en las prestaciones, inteligencia, calidad y facilidad de uso de su software (Android .vs. Windows mobile).

Además en la actualidad muy frecuentemente el coste de un software desarrollado a medida puede superar rápidamente el del hardware en el que se ejecuta. Esto es particularmente preciso en el caso de los PC. El precio habitual de estos dispositivos, normalmente inferior a los 1.000 €, apenas supone para una empresa el coste de un trabajador especializado durante un mes y este esfuerzo no permite abordar grandes aplicaciones.

Por otra parte, el desarrollo de software no es una tarea fácil. La complejidad actual de los sistemas informáticos hace a veces necesario el desarrollo de proyectos software de decenas de miles de líneas de código y el problema puede agravarse si existe la necesidad de optimizar los recursos de la máquina. Esto, no puede ser abordado directamente, empezando a programar sin más, particularmente si pretendemos desarrollar un software que sea mantenible. Es necesario analizar qué es lo que tenemos que hacer, cómo lo vamos a hacer, cómo se van a coordinar todas las personas que van a intervenir en el proyecto y cómo vamos a controlar el desarrollo del mismo de forma que al final obtengamos los resultados esperados.

El software es actualmente, dentro de cualquier sistema basado en el uso de ordenadores, el componente cuyo desarrollo presenta mayores problemas: es el más difícil de planificar, el que tiene mayor probabilidad de fracaso, y el que tiene menos posibilidades de que se cumplan las estimaciones de costes y tiempos iniciales. Por otra parte, la demanda de software (y también la complejidad del software que se demanda) aumentan continuamente, lo que aumenta la magnitud de estos problemas.

De todas formas, no hay que ser demasiado catastrofistas. El desarrollo de software es una actividad muy reciente (apenas tiene 60 años), si la comparamos con otras actividades de ingeniería (p.ej. la construcción de puentes o incluso la ingeniería eléctrica, de la que deriva la ingeniería de

hardware), y la disciplina que se encarga de establecer un orden en el desarrollo de sistemas de software (esto es, la Ingeniería del Software) es aún más reciente. Existen buenos métodos de desarrollo de software pero quizás el problema esté en que no están lo suficientemente difundidos o valorados. Sólo recientemente, estas técnicas están logrando una amplia aceptación.

1.2.- Evolución de la industria del software.

Hemos dicho que el software era el factor decisivo a la hora de elegir entre varias soluciones informáticas disponibles para un problema dado, pero esto no ha sido siempre así. En los primeros años de la informática, el hardware tenía una importancia mucho mayor que en la actualidad. Su coste era comparativamente mucho más alto y su capacidad de almacenamiento y procesamiento, junto con su fiabilidad, era lo que limitaba las prestaciones de un determinado producto. El software se consideraba como un simple añadido, a cuyo desarrollo se dedicaba poco esfuerzo y no se aplicaba ningún método sistemático. La programación era un arte de andar por casa, y el desarrollo de software se realizaba sin ninguna planificación. La mayoría del software se desarrollaba y era utilizado por la misma persona u organización. La misma persona lo escribía, lo ejecutaba y, si fallaba, lo depuraba. Debido a que la movilidad en el trabajo era baja, los ejecutivos estaban seguros de que esa persona estaría allí cuando se encontrara algún error. Debido a este entorno personalizado del software, el diseño era un proceso implícito, realizado en la mente de alguien y la documentación normalmente no existía.

En este contexto, las empresas de informática se dedicaron a mejorar las prestaciones del hardware, reduciendo los costes y el consumo de los equipos, y aumentando la velocidad de cálculo y la capacidad de almacenamiento. Para ello, tuvieron que dedicar grandes esfuerzos a investigación y aplicaron métodos de ingeniería industrial al análisis, diseño, fabricación y control de calidad de los componentes hardware. Como consecuencia de esto, el hardware se desarrolló rápidamente y la complejidad de los sistemas informáticos aumentó notablemente, necesitando de un software cada vez más complejo para su funcionamiento. Para enfrentarse a esta complejidad a finales de los 60 se desarrolla el concepto de programación estructurada que pasa a la industria a mediados de los 75. Por otro lado la aparición de casas comerciales dedicadas exclusivamente al desarrollo de software, amplió considerablemente el mercado de trabajo de los programadores. Con ello aumentó su movilidad laboral, y con la marcha de un trabajador desaparecían las posibilidades de mantener o modificar los programas que éste había desarrollado.

Al no utilizarse metodología alguna en el desarrollo del software, los programas contenían numerosos errores e inconsistencias, lo que obligaba a una depuración continua, incluso mucho después de haber sido entregados al cliente. Estas continuas modificaciones no hacían sino aumentar la inconsistencia de los programas, que se alejaban cada vez más de la corrección y se hacían prácticamente ininteligibles. Los costes se disparaban y frecuentemente era más rápido (y por tanto más barato) empezar de nuevo desde cero, desechando todo el trabajo anterior, que intentar adaptar un programa preexistente a un cambio de los requisitos. Sin embargo, los nuevos programas no estaban exentos de errores ni de futuras modificaciones, con lo que la situación volvía a ser la misma. Había comenzado la denominada crisis del software.

Para paliar, al menos en parte, estos problemas se amplia el enfoque de la programación estructurada al diseño. Se trata ahora de analizar los programas con un nivel de abstracción mayor en el que se piensa en el concepto de módulo como el componente básico de construcción. Pero incluso si esto ayudaba a mejorar los programas existía un problema previo que implicaba que podíamos estar haciendo un buen programa para resolver un problema equivocado. Es decir, previo

al diseño es imprescindible un análisis que nos defina que hay que hacer. Este análisis comenzó siendo una especificación narrativa de los requisitos pero pronto se introdujo también en esta etapa la idea de estructuración.

Hoy en día, la distribución de costes en el desarrollo de sistemas informáticos ha cambiado drásticamente. El software, en lugar del hardware, es el elemento principal del coste. Esto ha hecho que las miradas de los ejecutivos de las empresas se centren en el software y a que se formulen las siguientes preguntas:

- ¿ Por qué lleva tanto tiempo terminar los programas ?
- ¿ Por qué es tan elevado el coste ?
- ¿ Por qué no es posible encontrar todos los errores antes de entregar el software al cliente ?
- ¿ Por qué resulta tan difícil constatar el progreso conforme se desarrolla el software ?
- ¿Por qué se gasta tanto esfuerzo (tiempo y dinero) en el mantenimiento del software existente?

Estas y otras muchas preguntas son una manifestación del carácter del software y de la forma en que se desarrolla y han llevado a la aparición y la adopción paulatina de la ingeniería del software. Según Sommerville ésta puede definirse como la disciplina de la Ingeniería que concierne a todos los aspectos de la producción de software. En esta definición existen dos conceptos, uno el software, del que nos ocuparemos en este tema y otro, el proceso del que nos ocuparemos en los temas subsiguientes.

1.3.- El Software

Pressman nos propone la siguiente definición de software:

Software: (1) instrucciones de ordenador que cuando se ejecutan proporcionan la función y el comportamiento deseado, (2) estructuras de datos que facilitan a los programas manipular adecuadamente la información, y (3) documentos que describen la operación y el uso de los programas.

Por tanto, el software incluye no sólo los programas de ordenador, sino también las estructuras de datos que manejan esos programas y toda la documentación que debe acompañar al proceso de desarrollo, mantenimiento y uso de dichos programas.

1.3.1.- Características.

Según su definición, el software se diferencia de otros productos que los hombres puedan construir en que es, por su propia naturaleza lógico. En el desarrollo del hardware, el proceso creativo (análisis, diseño, construcción, prueba) se traduce finalmente en una forma material, en algo físico. Por el contrario, el software es inmaterial y por ello tiene unas características completamente distintas al hardware. Pressman nos cita las siguientes:

El software se desarrolla, no se fabrica en sentido estricto.

Existen similitudes entre el proceso de desarrollo del software y el de otros productos industriales, como el hardware. En ambos casos existen fases de análisis, diseño y desarrollo o

construcción, y la buena calidad del producto final se obtiene mediante un buen diseño. Sin embargo, en la fase de producción del hardware pueden producirse problemas que afecten a la calidad y que no existen, o son fácilmente evitables, en el caso del software

Por otro lado, en el caso de producción de hardware a gran escala, el coste del producto acaba dependiendo exclusivamente del coste de los materiales empleados y del propio proceso de producción, reduciéndose la repercusión en el coste de las fases de ingeniería previas. En el software, el desarrollo es una más de las labores de ingeniería, y la producción a gran o pequeña escala no influye en el impacto que tiene la ingeniería en el coste, al ser el producto inmaterial. Por otro lado, la replicación del producto (lo que sería equivalente a la producción del producto hardware) no presenta problemas técnicos, y no se requiere un control de calidad individualizado.

El diferente impacto que tiene la ingeniería en el desarrollo de productos hardware y software puede verse en el siguiente ejemplo:

	Ingeniería	Producción o Desarrollo	Coste unitario / 100 unidades	Coste unitario / 100.000 unidades
Hardware	1000	50 c.u.	60	50.01
Software	1000	2000	30	0.03

Tabla 1.1. Influencia de los costes de ingeniería en el coste total del producto

Los costes del software se encuentran en la ingeniería (incluyendo en ésta el desarrollo), y no en la producción, y es ahí donde hay que incidir para reducir el coste final del producto.

El software no se estropea.

Podemos comparar las curvas de índices de fallos del hardware y el software en función del tiempo. En el caso del hardware (figura 1.1), se tiene la llamada *curva de bañera*, que indica que el hardware presenta relativamente muchos fallos al principio de su vida. Estos fallos son debidos fundamentalmente a defectos de diseño o a la baja calidad inicial de la fase de producción. Una vez corregidos estos defectos, la tasa de fallos cae hasta un nivel estacionario y se mantiene así durante un cierto periodo de tiempo. Posteriormente, la tasa de fallos vuelve a incrementarse debido al deterioro de los componentes, que van siendo afectados por la suciedad, vibraciones y la influencia de muchos otros factores externos. Llegados a este punto, podemos sustituir los componentes defectuosos o todo el sistema por otros nuevos, y la tasa de fallos vuelve a situarse en el nivel estacionario.

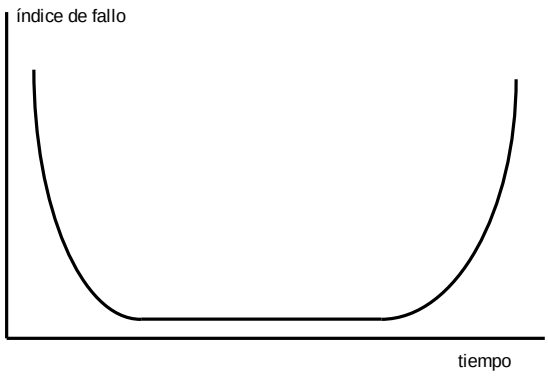

Figura. 1.1. Curva de fallos del hardware

El software no es susceptible a los factores externos que hacen que el hardware se estropee. Por tanto la curva debería seguir la forma de la figura 1.2. Inicialmente la tasa de fallos es alta, debido a la presencia de errores no detectados durante el desarrollo, los denominados errores latentes. Una vez corregidos estos errores, la tasa de fallos debería alcanzar el nivel estacionario y mantenerse ahí indefinidamente.

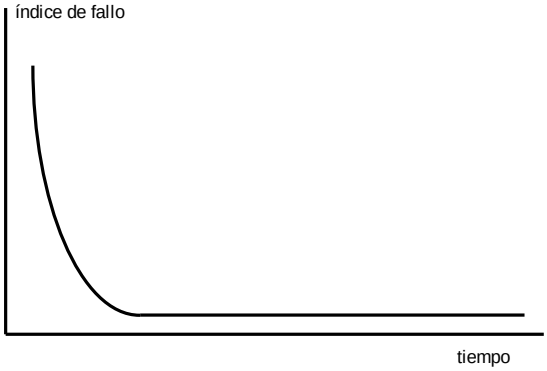

Figura 1.2. Curva ideal de fallos del software

Pero esto no es más que una simplificación del modelo real de fallos de un producto software. Durante su vida, el software sufre cambios, debidos al mantenimiento. El mantenimiento puede deberse a la corrección de errores latentes o a cambios en los requisitos iniciales del producto. Conforme se hacen cambios es bastante probable que se introduzcan nuevos errores, con lo que se producen picos en la curva de fallos.

Estos errores pueden corregirse, pero el efecto de los sucesivos cambios hace que el producto se aleje cada vez más de las especificaciones iniciales de acuerdo a las cuales fue desarrollado, conteniendo cada vez más errores latentes. Además, con mucha frecuencia se solicita - y se emprende - un nuevo cambio antes de haber conseguido corregir todos los errores producidos por el cambio anterior.

Por estas razones, el nivel estacionario que se consigue después de un cambio es algo superior al que el que había antes de efectuarlo (figura 1.3.), degradándose poco a poco el funcionamiento del sistema. Podemos decir entonces que el software no se estropea, pero se deteriora.

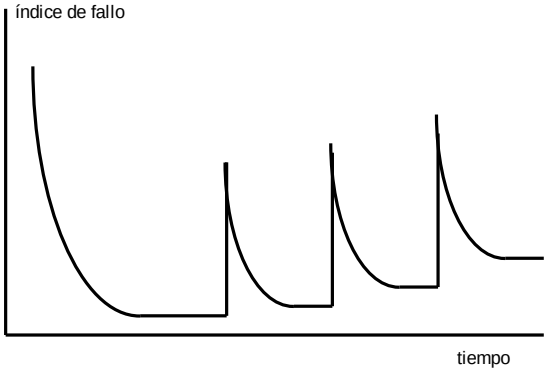

Figura 1.3. Curva real de fallos del software

Además, cuando un componente software se deteriora, no podemos sustituirlo por otro, como en el caso del hardware: no existen piezas de repuesto. Cada fallo del software indica un fallo

en el diseño o en el proceso mediante el cual se transformó el diseño en código máquina ejecutable. La solución no es sustituir el componente defectuoso por otro (que sería idéntico y contendría los mismos errores) sino un nuevo diseño y desarrollo del producto. Por tanto, el mantenimiento del software tiene una complejidad mucho mayor que el mantenimiento del hardware.

La mayoría del software se construye a medida.

El diseño de hardware se realiza, en gran medida, a base de componentes digitales existentes, cuyas características se comprueban en un catálogo y que han sido exhaustivamente probados por el fabricante y los usuarios anteriores. Estos componentes cumplen unas especificaciones claras y tienen unas interfaces definidas.

Actualmente el caso del software es distinto. No existen catálogos de componentes, y aunque determinados productos como sistemas operativos, editores, entornos de ventanas y bases de datos se venden en grandes ediciones, la mayoría del software se fabrica a medida, siendo en general la reutilización muy baja. Esta tendencia está cambiando en los últimos años con la aparición de repositorios de rutinas y procedimientos en la red, frameworks de desarrollo que ya integran soluciones a problemas comunes, y software con librerías específicas (de procesado de imagen, estadísticas, etc.). De todas formas la reutilización se está limitando al código, de forma que todavía no se ha extendido al diseño o al análisis.

El hecho de una baja reutilización tiene importantes consecuencias sobre las aplicaciones finales construidas:

- El impacto de los costes de ingeniería sobre el producto final se hace muy elevado, al dividirse entre un número de unidades producidas muy pequeño.
- La Productividad se reduce ya que un volumen importante de código, diseños, y cualquier otro elemento susceptible de reuso se construyen una y otra vez
- Se reduce la calidad al difuminarse entre distintas versiones el número de usuarios que verifica y valida un elemento.
- Se dificulta el mantenimiento y el soporte al consistir cada proyecto en islas independientes sin aspectos comunes que de existir podrían mantenerse por sustitución
- Se complica el control y la planificación de proyectos al tratar cada uno como un proyecto totalmente nuevo sin recurrir a elementos ya desarrollados en otros proyectos.

Se han hecho muchos intentos para conseguir aumentar el nivel de reutilización, todos ellos exigen comenzar con la identificación y creación de algún artefacto o activo reutilizable. Entendemos aquí por artefacto o activo cualquier elemento lógico fruto de algún proceso del desarrollo del software evitando así ceñirnos exclusivamente al código. Aún así los primeros ejemplos de dichos artefactos se centraron en el código y se han estado usando desde hace tiempo: son las bibliotecas. Durante los años sesenta se empezaron a desarrollar bibliotecas de subrutinas científicas, reutilizables en una amplia gama de aplicaciones científicas y de ingeniería. La mayor parte de los lenguajes modernos incluyen bibliotecas de este tipo, así como otras para facilitar la entrada / salida y más recientemente los entornos de ventanas. Sin embargo esta aproximación no siempre era posible y fallaba incluso para problemas de uso muy frecuente, como puede ser la búsqueda de un elemento en una estructura de datos. Los algoritmos disponibles para realizar estas

búsquedas eran dependientes de la organización interna de dichas estructuras y su fuerte variabilidad obligaba a reescribirlos una y otra vez, adaptándolos a cada situación particular.

Un nuevo intento de conseguir la reutilización se produjo con la utilización de técnicas de programación estructurada y modular. Sin embargo, se dedica por lo general poco esfuerzo al diseño de módulos lo suficientemente generales para ser reutilizables, y en todo caso, no se documentan ni se difunden todo lo que sería necesario para extender su uso, con lo que la tendencia habitual es diseñar y programar módulos muy semejantes una y otra vez. La programación estructurada permite diseñar programas con una estructura más clara, y que, por tanto, sean más fáciles de entender. Esta estructura interna más clara y estudiada, permite la reutilización de módulos dentro de los programas, o incluso dentro del proyecto que se está desarrollando, pero la reutilización de código en proyectos diferentes es muy baja.

Otra estrategia para conseguir la reutilización es el uso de técnicas orientadas a objetos, que permiten la programación por especialización. Los objetos, que encapsulan tanto datos como los procedimientos que los manejan - los métodos -, haciendo los detalles de implementación invisibles e irrelevantes a quien los usa, disponen de interfaces claras, los errores cometidos en su desarrollo pueden ser depurados sin que esto afecte a la corrección de otras partes del código y pueden ser heredados y reescritos parcialmente, haciendo posible su reutilización aún en situaciones no contempladas en el diseño inicial. Dentro de estas técnicas rápidamente ha aparecido la posibilidad de reusar otro tipo de artefactos, a través de los patrones se plantea la reutilización, no de código sino de diseños.

Actualmente organismos internacionales propugnan el reuso de artefactos como una buena política de empresa pero para poder llevarla a cabo es preciso una serie de actividades y la creación de los departamentos con la responsabilidad de llevarlas a cabo. Así es preciso que la empresa identifique sus artefactos reutilizables, difunda la información y obligue a uso enseñando a utilizar dichos activos. Para lograr estos propósitos con eficacia es preciso una reestructuración de la empresa que deberá dedicar recursos humanos orientados a analizar los artefactos de la empresa con los objetivos de detectar, difundir y obligar a su reuso.

1.3.2.- Atributos del software

El conjunto de atributos (Sommersville 2002) que se esperan de un software depende de su aplicación. Un sistema bancario o de comercio electrónico necesitará como principal atributo ser seguro; un juego debe mostrar, sobre todo, capacidad de respuesta y sencillez de manejo; y, por ejemplo, un software en tiempo real debe muy frecuentemente ser muy eficiente para aprovechar al máximo los recursos disponibles. El objetivo final debe ser crear un software que tenga en mayor o menor medida los siguientes atributos.

Mantenibilidad: El software debe escribirse de tal forma que pueda evolucionar para cumplir las necesidades de cambio de los clientes. Este es un atributo crítico ya que el cambio en el software es una consecuencia inevitable del cambio en sus entornos de funcionamiento.

Confiabilidad: La confiabilidad de un software implica diferentes aspectos como la seguridad, fiabilidad o protección. El software confiable no debe producir consecuencias económicas ni, por supuesto, físicas en caso de fallo.

Eficiencia: El software no debe malgastar recursos del sistema, como memoria o ciclos de procesamiento. Por lo tanto la eficiencia influye en los tiempos de respuesta, de procesamiento o en el uso de la memoria. La rápida evolución en el hardware y el rápido incremento de coste que este atributo supone hace que con frecuencia sea ignorado o menospreciado, excepto naturalmente, en aquellos casos que implique una prioridad.

Usabilidad: El software debe ser lo más sencillo posible de usar de forma que se adapte al usuario para el que está diseñado y no al revés. Esto afecta al desarrollo de los interface de usuario pero también a la documentación proporcionada.

1.3.3.- Aplicaciones del software.

El software puede aplicarse a numerosas situaciones del mundo real. En primer lugar, a todos aquellos problemas para los que se haya establecido un conjunto específico de acciones que lleven a su resolución (esto es, un algoritmo). En estos casos, utilizaremos lenguajes de programación procedimentales para implementar estos algoritmos. También puede aplicarse a situaciones en las que el problema puede describirse formalmente, por lo general en forma recursiva. En estos casos no necesitamos describir el método de resolución, es decir *cómo se resuelve el problema*, sino que bastará con describir el problema en sí, indicando cuál es la solución deseada, y utilizaremos lenguajes declarativos para ello. También puede aplicarse a problemas que los humanos resolvemos utilizando multitud de reglas heurísticas posiblemente contradictorias, para lo cual utilizaremos un sistema experto e incluso para problemas de los cuales no tenemos una idea clara de cómo se resuelven, pero de los que conocemos cuál es la solución apropiada para algunos ejemplos de los datos de entrada. En este caso utilizaremos redes neuronales.

En cualquier caso, es difícil establecer categorías genéricas significativas para las aplicaciones del software. Conforme aumenta la complejidad del mismo se hace más complicado establecer compartimentos nítidamente separados. No obstante Pressman nos ofrece la siguiente clasificación:

Software de sistemas.

El software de sistemas es una colección de programas escritos para servir a otros programas. Algunos programas de sistemas (como los compiladores, editores y utilerías para la administración de archivos) procesan estructuras de información complejas pero determinadas. Otras aplicaciones de sistemas (por ejemplo, componentes del sistema operativo, controladores, software de red, procesadores para telecomunicaciones) procesan datos indeterminados. En cada caso, el área de software de sistemas se caracteriza por una interacción muy intensa con el hardware de la computadora; utilización por múltiples usuarios; operación concurrente que requiere la gestión de itinerarios, de compartición de recursos, y de procesos sofisticados; estructuras de datos complejas y múltiples interfaces externas.

Software de aplicación.

El software de aplicación consiste en programas independientes que resuelven una necesidad de negocios específica. Las aplicaciones en esta área procesan datos empresariales o técnicos de

forma que facilitan las operaciones de negocios o la toma de decisiones técnicas o de gestión. Además del procesamiento de datos convencional, el software de aplicación se utiliza para controlar las funciones de negocios en tiempo real (por ejemplo, el procesamiento de transacciones en los puntos de venta y el control de procesos de manufactura en tiempo real.)

Software científico y de ingeniería.

El software científico y de ingeniería, que se caracterizaba por algoritmos "devoradores de números", abarca desde la astronomía hasta la vulcanología, desde el análisis de la tensión automotriz hasta la dinámica orbital de los transbordadores espaciales, y desde la biología molecular hasta la manufactura automatizada. Sin embargo, las aplicaciones modernas dentro del área científica y de ingeniería se alejan en la actualidad de los algoritmos numéricos convencionales. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas han comenzado a tomar características de software en tiempo real e incluso de software de sistemas.

Software empotrado.

El software empotrado reside dentro de la memoria de sólo lectura del sistema y con él se implementan y controlan características y funciones para el usuario final y el sistema mismo. El software incrustado puede desempeñar funciones limitadas y curiosas (como el control del teclado de un horno de microondas) o proporcionar capacidades de control y funcionamiento significativas (por ejemplo, las funciones digitales de un automóvil, como el control de combustible, el despliegue de datos en el tablero, los sistemas de frenado, etcétera).

Software de línea de productos.

El software de línea de productos, diseñado para proporcionar una capacidad específica y la utilización de muchos clientes diferentes, se puede enfocar en un nicho de mercado limitado (como en los productos para el control de inventarios) o dirigirse hacia los mercados masivos (por ejemplo, aplicaciones de procesadores de palabras, hojas de cálculo, gráficas por computadora, multimedia, entretenimiento, manejo de bases de datos, administración de personal y finanzas en los negocios).

Aplicaciones basadas en Web.

Las "WebApps" engloban un espectro amplio de aplicaciones. En su forma más simple, las WebApps son apenas un poco más que un conjunto de archivos de hipertexto ligados que presenta información mediante texto y algunas gráficas. Sin embargo, a medida que el comercio electrónico y las aplicaciones B2B adquieren mayor importancia, las WebApps evolucionan hacia ambientes computacionales sofisticados que no sólo proporcionan características, funciones de cómputo y contenidos independientes al usuario final, sino que están integradas con bases de datos corporativas y aplicaciones de negocios.

Software de inteligencia artificial.

Este software utiliza algoritmos no numéricos en la resolución de problemas complejos que es imposible abordar por medio de un análisis directo. Las aplicaciones dentro de esta área incluyen la robótica, los sistemas expertos, el reconocimiento de patrones (imagen y voz), las redes neuronales artificiales, la comprobación de teoremas y los juegos en computadora.

Como vemos, el software permite aplicaciones muy diversas, pero en todas ellas podemos encontrar algo en común: el objetivo es que el software desempeñe una determinada función, y además, debe hacerlo cumpliendo una serie de restricciones. Estas pueden ser muy variadas: corrección, fiabilidad, respuesta en un tiempo determinado, facilidad de uso, bajo coste, etc., pero siempre existen y no podemos olvidarnos de ellas a la hora de desarrollar el software.

1.3.4.- El software heredado

El software heredado hace referencia a sistemas desarrollados hace décadas y que han sido modificados de forma continua para cumplir los requisitos de los cambios en los negocios y en las plataformas de cómputo. Tales sistemas resultan muy caros de mantener pero su evolución además de costosa supone un claro riesgo para la empresa cuyo negocio se sostiene sobre la base de dicho software. El software heredado se caracteriza por tanto por su longevidad y por ser crítico para los negocios.

La longevidad de este software va normalmente de la mano de la falta de calidad. Esto no implica que su desarrollo haya sido descuidado sino simplemente que su proceso de desarrollo se ha quedado desfasado. En la mayoría de los casos los criterios de calidad ni siquiera habían sido definidos y con frecuencia la ingeniería del software se encontraba en sus inicios y muy alejada de los principios que ahora se manejan. Como consecuencia sus diseños son con frecuencia imposibles de extender, su código es complicado, su documentación escasa o inexistente, sus pruebas nunca fueron archivadas y su historial de cambios ha sido seguido con pobreza. La lista podría extenderse aún más, sin embargo, dichos sistemas continúan sosteniendo a la organización [Dayani-Fard, 1999; Liu, 1998].

Ante esta situación la mejor opción es no actuar, al menos mientras la situación no exija un cambio significativo y este tiende a darse debido a diferentes factores que fuerzan, en general, a todo el software a evolucionar.

- El software debe adaptarse para satisfacer los avances en las tecnologías de cómputo
- El software debe mejorarse para atender nuevos requerimientos
- Debe extenderse para permitirle relacionarse con nuevos sistemas
- Debe rediseñarse para hacerlo viable dentro de un ambiente en red.

Como resultas más tarde o más temprano el software tiende a evolucionar y esto se ha convertido en una fuente de interés. La evolución del software se entiende en la actualidad como un aspecto más del proceso de desarrollo, se puede plantear como el único objetivo en un proyecto y se proponen trabajos que tratan de desarrollar leyes que sean verificadas en la evolución de cualquier software [Lehman, 1997]. En el caso del software antiguo se plantea además otra necesidad abordada por la reingeniería del software. En este proceso, que será abordado posteriormente, se parte de una aplicación operativa para extraer de ella una visión de alto nivel que utilizar en el rediseño de la aplicación utilizando los principios de la ingeniería.

1.3.5.- Principales problemas asociados a la producción del software.

Hemos hablado de una *crisis del software*. Sin embargo, por crisis entendemos normalmente un estado pasajero de inestabilidad, que tiene como resultado un cambio de estado del sistema o una vuelta al estado inicial, en caso de que se tomen las medidas para superarla. Teniendo en cuenta esto, el software, más que padecer una crisis podríamos decir que padece una enfermedad crónica. Los problemas que surgieron cuando se empezó a desarrollar software de una cierta complejidad siguen existiendo actualmente, sin que se haya avanzado mucho en los intentos de solucionarlos.

Estos problemas son causados por las propias características del software y por los errores cometidos por quienes intervienen en su producción. Entre ellos podemos citar:

La planificación y la estimación de costes son muy imprecisas.

A la hora de abordar un proyecto de una cierta complejidad, sea en el ámbito que sea, es frecuente que surjan imprevistos que no estaban recogidos en la planificación inicial, y como consecuencia de estos imprevistos se producirá una desviación en los costes del proyecto. Sin embargo, en el desarrollo de software lo más frecuente es que la planificación sea prácticamente inexistente, y que nunca se revise durante el desarrollo del proyecto. Sin una planificación detallada es totalmente imposible hacer una estimación de costes que tenga alguna posibilidad de cumplirse, y tampoco se pueden identificar las tareas conflictivas que pueden desviarnos de los costes previstos. Entre las causas de este problema podemos citar:

- No se recogen datos sobre el desarrollo de proyectos anteriores, con lo que no se acumula experiencia que pueda ser utilizada en la planificación de nuevos proyectos.

- Los gestores de los proyectos no están especializados en la producción de software. Tradicionalmente, los responsables del desarrollo del software han sido ejecutivos de nivel medio y alto sin conocimientos de informática, siguiendo el principio de 'Un buen gestor puede gestionar cualquier proyecto'. Esto es cierto, pero no cabe duda de que también es necesario conocer las características específicas del software, aprender las técnicas que se aplican en su desarrollo y conocer una tecnología que evoluciona continuamente.

La productividad es baja.

Los proyectos software tienen, por lo general, una duración mucho mayor a la esperada. Como consecuencia de esto los costes se disparan y la productividad y los beneficios disminuyen. Uno de los factores que influyen en esto, es la falta de unos propósitos claros o realistas a la hora de comenzar el proyecto. La mayoría del software se desarrolla a partir de una especificaciones ambiguas o incorrectas, y no existe apenas comunicación con el cliente hasta la entrega del producto. Debido a esto son muy frecuentes las modificaciones de las especificaciones sobre la marcha o los cambios de última hora, después de la entrega al cliente. No se realiza un estudio detallado del impacto de estos cambios y la complejidad interna de las aplicaciones crece hasta que se hacen virtualmente imposibles de mantener y cada nueva modificación, por pequeña que sea, es más costosa, y puede provocar el fallo de todo el sistema.

Debido a la falta de documentación sobre cómo se ha desarrollado el producto o a que las sucesivas modificaciones - también indocumentadas - han desvirtuado totalmente el diseño inicial, el

mantenimiento de software puede llegar a ser una tarea imposible de realizar, pudiendo llevar más tiempo el realizar una modificación sobre el programa ya escrito que analizarlo y desarrollarlo entero de nuevo.

La calidad es mala.

Como consecuencia de que las especificaciones son ambiguas o incluso incorrectas, y de que no se realizan pruebas exhaustivas, el software contiene numerosos errores cuando se entrega al cliente. Estos errores producen un fuerte incremento de costes durante el mantenimiento del producto, cuando ya se esperaba que el proyecto estuviese acabado. Sólo recientemente se ha empezado a tener en cuenta la importancia de la prueba sistemática y completa, y han empezado a surgir conceptos como la fiabilidad y la garantía de calidad.

El cliente queda insatisfecho.

Debido al poco tiempo e interés que se dedican al análisis de requisitos y a la especificación del proyecto, a la falta de comunicación durante el desarrollo y a la existencia de numerosos errores en el producto que se entrega, los clientes suelen quedar muy poco satisfechos de los resultados. Consecuencia de esto es que las aplicaciones tengan que ser diseñadas y desarrolladas de nuevo, que nunca lleguen a utilizarse o que se produzca con frecuencia un cambio de proveedor a la hora de abordar un nuevo proyecto.

1.4.- Bibliografía

[Dayani, 1999] Dayani-Fard, H. et al., "legacy Software Systems: Issues, Progress, and Challenges", IBM Technical Report: TR-74. 165-k, abril de 1999, disponible en http://www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html.

[Lehman, 1997] Lehman, M. et al. "Metrics and Laws of Software Evolutions-The Nineties Views", en Procceedings of the 4th International Software Metrics Symposium(METRICS´97), IEEE, 1997, disponible en http://www.ece.utexas.edu/~perry/work/papers/feast1.pdf

[Liu, 1998] Liu, K. et al., "Reoport on the First SEBPC workshop on Legacy Systems", Durham University, febrero de 1998, disponible en http://www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html.

Roger S Pressman, 2005; "Ingeniería del Software. Un enfoque práctico. 6ª Edicion". Ed. Mc Graw Hill, España. ISBN: 970-10-5473-3