

Tema 1. El producto	1
1.1.- Introducción.....	1
1.2.- Evolución de la industria del software.....	2
1.3.- El Sofware	3
1.3.1.- Características.....	3
1.3.2.- Atributos del software	7
1.3.3.- Aplicaciones del software.....	8
1.3.4.- El software heredado	10
1.3.5.- Principales problemas asociados a la producción del software.	11
1.4.- Bibliografia.....	12

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, desechar 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.

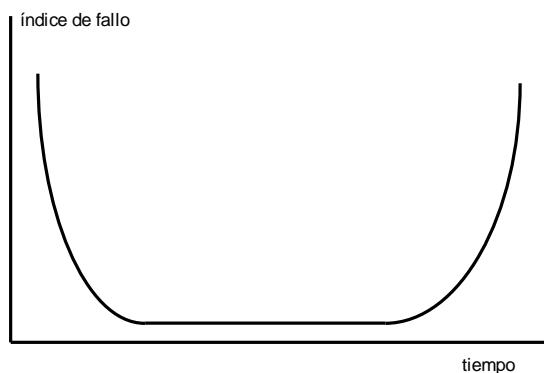


Figura. 1.1. Curva de fallos del hardware

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.

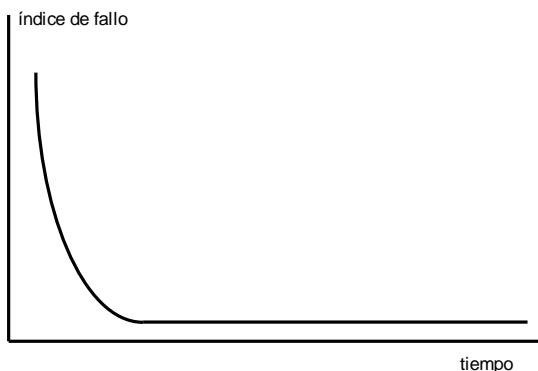


Figura 1.2. Curva ideal de fallos del software

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.

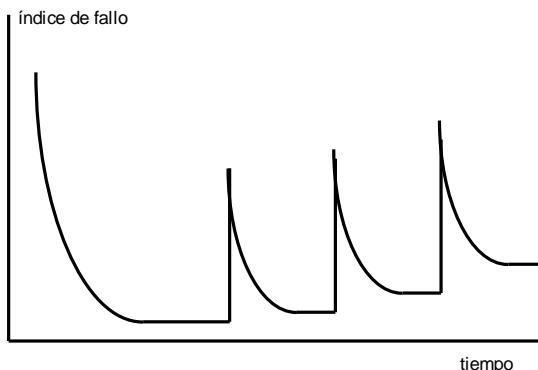


Figura 1.3. Curva real de fallos del software

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 (Sommerville 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 comportamientos 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 Proceedings 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., “Report 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. 6^a Edición”. Ed. Mc Graw Hill, España. ISBN: 970-10-5473-3

Tema 2. Ingeniería de software. El proceso

Tema 2. Ingeniería de software. El proceso	i
2.1.- Definición de Ingeniería del Software.	15
2.2.- Los procesos para la construcción del software.....	16
2.2.1.- La norma IEEE 1074 [Piattini 2003]	18
2.2.2.- Norma ISO 12207 – 1 [Piattini 1996].....	21
2.2.3.- La norma ISO/IEC TR 15504-2 [Piattini 2003]......	25
2.3.- Evaluación del proceso software.....	27
2.3.1.- Integración del Modelo de Capacidad de Madurez (CMMI).....	28

Índice de Figuras

Figura 2.1.- Secciones lógicas para los procesos en el estándar IEEE 1074.....	19
Figura 2.2.- Agrupación de procesos de la norma ISO 12207 - 1	21
Figura 2.3.- Procesos de la ISO/IEC 15504-2 y su alineamiento con la ISO 12207-1	25
Figura 2.4. Procesos de la norma ISO/IEC 15504-2 (rev. 2003)	27
Figura 2.5. Relación con los procesos de evaluación.....	28
Figura 2.6. Áreas de proceso por categoría para el CMMI continuo.	29
Figura 2.7. PAs por nivel de madurez del modelo CMMI discreto.	31

2.1.- Definición de Ingeniería del Software.

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

Los problemas tradicionales en el desarrollo de software no van a desaparecer de la noche a la mañana, pero identificarlos y conocer sus causas es el único método que nos puede llevar hacia su solución.

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

Una de las primeras definiciones que se dio de la ingeniería del software propuesta por Fritz Bauer es la siguiente:

Ingeniería del software es el establecimiento y uso de principios de ingeniería robustos, orientados a obtener software económico, que sea fiable y funcione de manera eficiente sobre máquinas reales. [Nau69]

Esta definición no es más que un establecimiento de principios para resolver la crisis del software de la que hemos hablado y para el que la definición propone a la ingeniería del software como la respuesta definitiva.

El IEEE [IEEE93] ha desarrollado una definición más completa según la cual:

Ingeniería del software: La aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software.

En esta definición está implícita la existencia de fases por las que se debe hacer pasar al software de manera sistemática, y lo que es más complicado, de forma cuantificable. Es decir, la ingeniería del software nos debe proporcionar una metodología de desarrollo de software que nos precise en cada momento qué pasos debemos seguir para construir un software económico y de calidad. Aún más nos debe proporcionar la forma de averiguar en qué parte del proceso estamos y cuán bien lo estamos haciendo.

La primera aproximación a esta idea se hace a través de una serie de paradigmas que tratan de describir las fases que debe atravesar el proceso de desarrollo de software desde que éste es concebido hasta que deja de usarse. A dichos paradigmas se les denomina CICLOS DE VIDA DEL SOFTWARE y serán analizados posteriormente. Esta aproximación surge del análisis del proceso de Software desde la globalidad al detalle con esta perspectiva también se plantean otras ideas como las de proceso o estándares que no buscan una descripción detallada de las actividades si no su identificación. Pero esta perspectiva no es la única posible, se plantean así las metodologías que tienen históricamente una perspectiva menos amplia del problema pero mayor nivel de detalle en la descripción de las actividades.

En cualquier caso, las fronteras entre estos conceptos se mezclan con frecuencia. En general es interesante el planteamiento que propone Piattini al respecto utilizando la definición de ciclo de

vida del IEEE como una serie de procesos que se deben realizar sin definir el detalle de cómo llevarlos a cabo. Posteriormente introduce el concepto de metodología como las estrategias que definen con precisión los pasos que se deben seguir en cada momento. Aclara, sin embargo, que no existe una metodología estándar que pueda aplicarse al desarrollo de cualquier proyecto software sino una serie de ellas que además, cada empresa debería refinar para adecuarla a su particular idiosincrasia.

En cualquier caso debe quedar claro que la ingeniería del software busca formalizar la secuencia de pasos que hay que dar para completar con éxito el desarrollo de un software de calidad y que, aunque en este sentido no está todo dicho, si existen una serie de normas y estrategias que se deben seguir.

A lo largo de su historia se han propuesto distintos ciclos de vida del software para los que se definen distintas fases e interrelaciones entre ellas. En este proceso han surgido distintos términos como metodología, proyectos, ciclo de vida, procesos, fases, actividades, tareas, procedimientos, métodos, técnicas o herramientas que, según que autor sigamos, pueden ser usados con interpretaciones ligeramente distintas, a veces algunos términos son usados como sinónimos y en otros casos se les dan acepciones distintas. Todo esto es fruto de que el proceso de sistematización todavía se encuentra en fase de desarrollo y conviene tenerlo muy presente cuando leamos bibliografía de distintos autores.

Aunque a continuación intentaremos clarificar estos términos no deben entenderse como definiciones formales aceptadas por todo el mundo sino como un intento de establecer una jerarquía útil entre los distintos conceptos.

Las herramientas son el concepto más claro y obvio están constituidas por cualquier software que proporcione soporte automático a cualquier actividad relacionada con la ingeniería del software. Con frecuencia las herramientas se centran en algún aspecto concreto y facilitan la ejecución de una o varias técnicas de forma que es preciso utilizar varias para abarcar todas las actividades del desarrollo del software.

Las técnicas se centran principalmente en un conjunto de gráficos con apoyos textuales formales y determinan el formato de los productos resultantes de cada tarea. Una idea importante a tener presente es que la misma técnica puede usarse con distintos objetivos.

Los métodos y procedimientos determinan el modo en el que se aplican las técnicas y el detalle de los elementos de los que se debe partir y los productos resultantes de aplicar dicho método o procedimiento. Indican cómo construir técnicamente el software, y abarcan una amplia serie de tareas que incluyen la planificación y estimación de proyectos, el análisis de requisitos, el diseño de estructuras de datos, programas y procedimientos, la codificación, las pruebas y el mantenimiento. Los métodos introducen frecuentemente una notación específica para la tarea en cuestión y una serie de criterios de calidad. Según los autores podemos encontrar que un método engloba a procedimientos o viceversa o bien encontrar que se usan como sinónimos.

Metodologías, ciclos de vida y proceso del software. Estos son los conceptos más amplios que nos vamos a encontrar. De nuevo y según el caso se incluyen unos a otros o se utilizan como sinónimos. En todos los casos estos conceptos precisan una serie de fases por las que se ha de pasar en el desarrollo del software y según los autores cada uno de estos conceptos incluye más o menos precisión en la definición de dichas fases.

2.2.- Los procesos para la construcción del software

Como hemos discutido con anterioridad la construcción del software incluye una serie de actividades que comienzan a estandarizarse. En esta sección empezaremos por una visión simple

propuesta por Pressman de las fases técnicas inexcusables para la construcción del software. Posteriormente veremos un par de normas que amplían considerablemente esta visión, la IEEE 1074 y la ISO 12207-1. En ambos casos se pretende detectar todas las actividades precisas para la construcción del software y veremos, a través de su evolución histórica como la lista tiende a ampliarse. Estas listas no sólo describen actividades directamente orientadas a la construcción del software sino también a dar soporte a tal actividad y a las empresas dedicadas a tal empeño.

Posteriormente analizaremos distintas estrategias que, centradas en el desarrollo del software, organizan las tareas implicadas en dicha actividad proponiendo distintos paradigmas de ejecución y por tanto de desarrollo del software. Tales paradigmas se conocen como los ciclos de vida del software.

Pressman (5 ed pp.15-16) entiende la construcción del software como un proceso dividido en tres fases en cada una de las cuales se responde a alguna de las preguntas que un proceso de ingeniería debe plantear y resolver:

- ¿Cuál es el problema a resolver?
- ¿Cuáles son las características de la entidad que se utiliza para resolver este problema?
- ¿Cómo se realizará la entidad?
- ¿Qué enfoque se va a utilizar para no contemplar los errores que se cometieron en el diseño y en la construcción de la entidad?
- ¿Cómo se sostendrá la entidad cuando los usuarios soliciten correcciones, adaptaciones y mejoras de la entidad?

Estas fases serían la de definición, la de desarrollo y la de mantenimiento

Fase de definición.

La fase de definición se centra en el **qué**. Durante esta fase, se intenta identificar: ¿qué información es la que tiene que ser procesada?; ¿qué función y rendimiento son los que se esperan?; ¿qué restricciones de diseño existen?; ¿qué interfaces deben utilizarse?; ¿qué sistema operativo y soporte hardware van a ser utilizados?; ¿qué criterios de validación se necesitan para conseguir que el sistema final sea correcto?

En esta fase se pueden identificar dos actividades: *Análisis del sistema* que define el papel de cada elemento relacionado con el sistema informático que se pretende desarrollar, precisando cuál es el papel del software dentro de ese sistema. *Análisis de requisitos del software* que proporciona el ámbito del software, su relación con el resto de componentes del sistema, pero antes de empezar a desarrollar es necesario hacer una definición más detallada de la función del software. La comprensión del producto a desarrollar, alcanzado en estas actividades, es básica para la realización de una buena *Planificación*, con la que organizar las tareas que se llevarán a cabo en la realización del proyecto.

Existen dos formas de realizar el análisis y refinamiento de los requisitos del software. Por una parte, se puede hacer un análisis formal del ámbito de la información para establecer modelos del flujo y la estructura de la información. Luego se amplían unos modelos para convertirlos en una especificación del software. La otra alternativa consiste en construir un prototipo del software, que será evaluado por el cliente para intentar consolidar los requisitos. Los requisitos de rendimiento y las limitaciones de recursos se traducen en directivas para la fase de diseño.

El análisis y definición de los requisitos es una tarea que debe llevarse a cabo conjuntamente por el desarrollador de software y por el cliente. La especificación de requisitos del software es el documento que se produce como resultado de esta etapa.

Fase de desarrollo

Durante el desarrollo un ingeniero de software intenta definir cómo han de diseñarse las estructuras de datos, cómo ha de implementarse la función dentro de una arquitectura software, cómo han de implementarse los detalles procedimentales, cómo han de caracterizarse los interfaces, cómo ha de traducirse el diseño en un lenguaje de programación y cómo ha de realizarse la prueba. En este caso se definen también tres actividades que serían el diseño del software, la codificación y las pruebas.

Fase de mantenimiento

Esta fase se centra en el cambio asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software y a cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente. En esta fase se definen las siguientes actividades

Corrección: asociada a cambios debidos a errores detectados por el cliente en la aplicación cuando ésta ya se encuentra operativa. *Adaptación*: cuando se introducen modificaciones en el programa original para adaptarlo a cambios en su entorno original. *Mejora*: en esta actividad el usuario descubre funcionalidades adicionales que quiere incorporar y que llevan al sistema más allá de sus especificaciones iniciales y la *Prevención* que busca mejorar el programa a fin de que sea más fácil mejorarlo, adaptarlo o corregir sus defectos.

Además estas actividades desarrolladas a lo largo de las tres fases se complementan con un número de actividades protectoras. Seguimiento y control del proyecto de software, revisiones técnicas formales, garantía de calidad del software, gestión de configuración del software, preparación y producción de documentos, gestión de reutilización, mediciones y gestión de riesgos.

2.2.1.- La norma IEEE 1074 [Piattini 2003]

Este estándar proporciona el conjunto de actividades que constituyen los procesos que son obligatorios para el desarrollo y mantenimiento de software. El estándar está organizado en 17 procesos, que comprenden un total de 65 actividades.

Los procesos se dividen en cuatro secciones lógicas, como se muestra en la Figura 2.2.

1. Proceso de Modelo del Ciclo de Vida Software (véase apartados 1.3.2 en adelante). Existen muchas variables que afectan a la selección por parte de una empresa de un modelo de ciclo de vida software, como por ejemplo el tipo de producto (interactivo, batch, procesamiento de transacciones, etc.), restricciones. Aunque el estándar no establece ni define un ciclo de vida específico o sus metodologías subyacentes, si requiere que se seleccione y utilice un modelo de ciclo de vida.

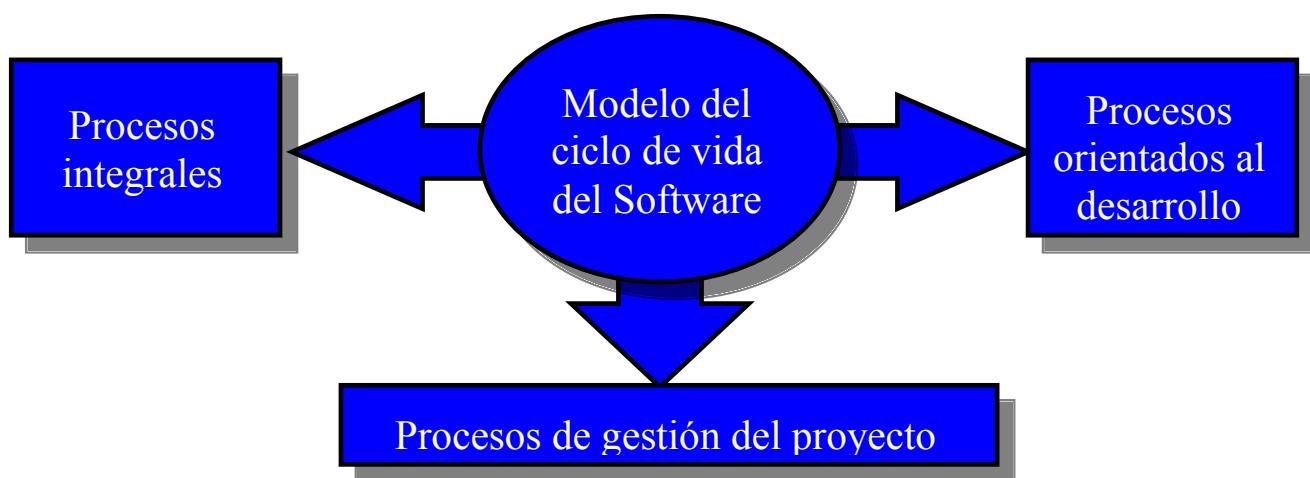


Figura 2.1.- Secciones lógicas para los procesos en el estándar IEEE 1074

Este proceso proporciona las actividades que se necesitan para identificar los modelos de ciclo de vida software candidatos y para seleccionar aquel modelo que se vaya a utilizar. Es posible que el mismo modelo de ciclo de vida se pueda aplicar a diferentes proyectos (por ejemplo, cascada); sin embargo aunque todos los proyectos sigan el modelo en cascada, cada proyecto deberá tener su propio ciclo de vida software en función del conjunto de actividades concretas que se vayan a realizar.

2. Procesos de Gestión del Proyecto. Son los procesos que inician, supervisan y controlan los proyectos software a lo largo del ciclo de vida software. Está formado por el Proceso de Iniciación del Proyecto, el Proceso de Supervisión y Control del Proyecto, y el Proceso de Gestión de la Calidad Software

- El Proceso de Iniciación del Proyecto consiste en aquellas actividades que crean y mantienen el marco del proyecto. Es decir, se crea el ciclo de vida software del proyecto y se establecen los planes para gestionar el proyecto.
- El Proceso de Supervisión y Control del Proyecto es un proceso iterativo de seguimiento, informes y gestión de costes, calendarios, problemas y rendimiento de un proyecto a lo largo de su ciclo de vida. El progreso de un proyecto se revisa y mide en los hitos establecidos en el plan del proyecto software,
- El Proceso de Gestión de la Calidad Software se utiliza para tratar la planificación y administración del programa de Aseguramiento de la Calidad Software. Además, trata la satisfacción del cliente y los programas internos de mejora de la calidad.

Las actividades dentro del Proceso de Supervisión y Control del Proyecto, y del Proceso de Gestión de la Calidad Software, se realizan durante toda la vida del proyecto para asegurar el nivel apropiado de gestión del proyecto y de cumplimiento con las actividades obligatorias.

3. Procesos Orientados al Desarrollo. Comprenden los procesos que se realizan antes (Procesos de Pre-Desarrollo), durante (Procesos de Desarrollo) y después (Procesos de Post-Desarrollo) del desarrollo software.

Los Procesos de Pre-Desarrollo incluyen el Proceso de Exploración del Concepto y el Proceso de Asignación del Sistema.

- El Proceso de Exploración del Concepto examina los requisitos a nivel del sistema y produce un Informe de Necesidades que inicia el Proceso de Asignación del Sistema o el Proceso de Requisitos. Incluye la identificación de una idea o necesidad, su evaluación y refinamiento, y, una vez obtenidos los límites, la generación de un Informe de Necesidades para desarrollar el sistema.
- El Proceso de Asignación del Sistema realiza la asignación de requisitos al software y al hardware. Este proceso no se aplica cuando el proyecto sólo requiere desarrollo software.

Los Procesos de Desarrollo incluyen el Proceso de Requisitos, el Proceso de Diseño y el Proceso de Implementación.

- El Proceso de Requisitos incluye aquellas actividades dirigidas al desarrollo de los requisitos software. En el desarrollo de un sistema que contenga tanto componentes hardware como software, el Proceso de Requisitos sigue al de desarrollo de los requisitos totales del sistema y a la asignación funcional de

dichos requisitos del sistema al hardware y al software.

- Durante el Proceso de Diseño se toman las principales decisiones para determinar la estructura del sistema. El objetivo del Proceso de Diseño es desarrollar una representación coherente y bien organizada del sistema software que cumpla los Requisitos Software. El Proceso de Diseño proyecta el ‘qué hacer’ de las especificaciones de requisitos en el ‘cómo hacerlo’ de las especificaciones de diseño. El diseño arquitectónico se centra en las funciones y estructura de los componentes software que comprenden el sistema software. El diseño detallado se centra en las estructuras de datos y algoritmos que se utilizan dentro de cada componente software.
- El Proceso de Implementación transforma la representación del diseño detallado de un producto software en una conversión del lenguaje de programación. Este proceso produce el código fuente, la base de datos y la documentación que constituye la manifestación física del diseño. Además, se integran el código y la base de datos. La salida de este proceso debe ser el objeto de todas las pruebas y validaciones subsiguientes.

Los Procesos de Post-Desarrollo incluyen el Proceso de Instalación, el Proceso de Explotación y Soporte, el Proceso de Mantenimiento y el Proceso de Retirada.

- El Proceso de Instalación consiste en el transporte e instalación de un sistema software desde el entorno de desarrollo hasta el entorno objeto. Incluye las modificaciones de software necesarias, la verificación en el entorno objeto y la aceptación del cliente. Durante este proceso, el software a entregar se instala, verifica y supervisa. Este esfuerzo culmina con la aceptación formal del cliente.
- El Proceso de Explotación y Soporte implica la operación del usuario del sistema y el soporte continuo. El soporte incluye proporcionar asistencia técnica, consulta del usuario y registro de las peticiones de soporte del usuario.
- El Proceso de Mantenimiento está relacionado con la resolución de los errores y fallos del software.
- El Proceso de Retirada implica la eliminación de un sistema existente de su soporte o uso activo por cese de su explotación o soporte, o bien por su sustitución por un nuevo sistema o una nueva versión actualizada del sistema existente.

4. Procesos Integrales. Son aquellos procesos que se necesitan para completar con éxito las actividades de un proyecto. Estos procesos se utilizan para asegurar la terminación y calidad de las funciones del proyecto. Está formado por el Proceso de Verificación y Validación, el Proceso de Gestión de la Configuración Software, el Proceso de Desarrollo de la Documentación y el Proceso de Formación.

- El Proceso de Verificación y Validación incluye la planificación y realización de las tareas de Verificación (revisiones, auditorías de configuración y auditorías de calidad) y de Validación (todas las fases de pruebas) que se realizan a lo largo del ciclo de vida software para asegurar que se satisfacen todos los requisitos.
- El Proceso de Gestión de Configuración Software identifica los elementos en un proyecto de desarrollo software y proporciona el control de los elementos identificados y la generación de informes de estado para que la dirección tenga visibilidad del ciclo de vida software.

- El Proceso de Desarrollo de la Documentación es el conjunto de actividades que planifican, diseñan, implementan, editan, producen, distribuyen y mantienen aquellos documentos necesitados por los desarrolladores y usuarios.
- El Proceso de Formación es esencial para los desarrolladores, plantilla de soporte técnico y clientes. El desarrollo de productos software de calidad depende ampliamente del conocimiento y habilidad de las personas.

2.2.2.- Norma ISO 12207 – 1 [Piattini 1996]

La norma ISO 12207 –1 define una serie de actividades que se realizan en la construcción del software agrupadas en cinco procesos principales, ocho de soporte y cuatro más de la organización. La norma no fomenta sin embargo ningún modelo concreto de ciclo de vida, gestión del software o método de ingeniería ni prescribe ni cómo realizar las actividades ni cómo organizarlas.

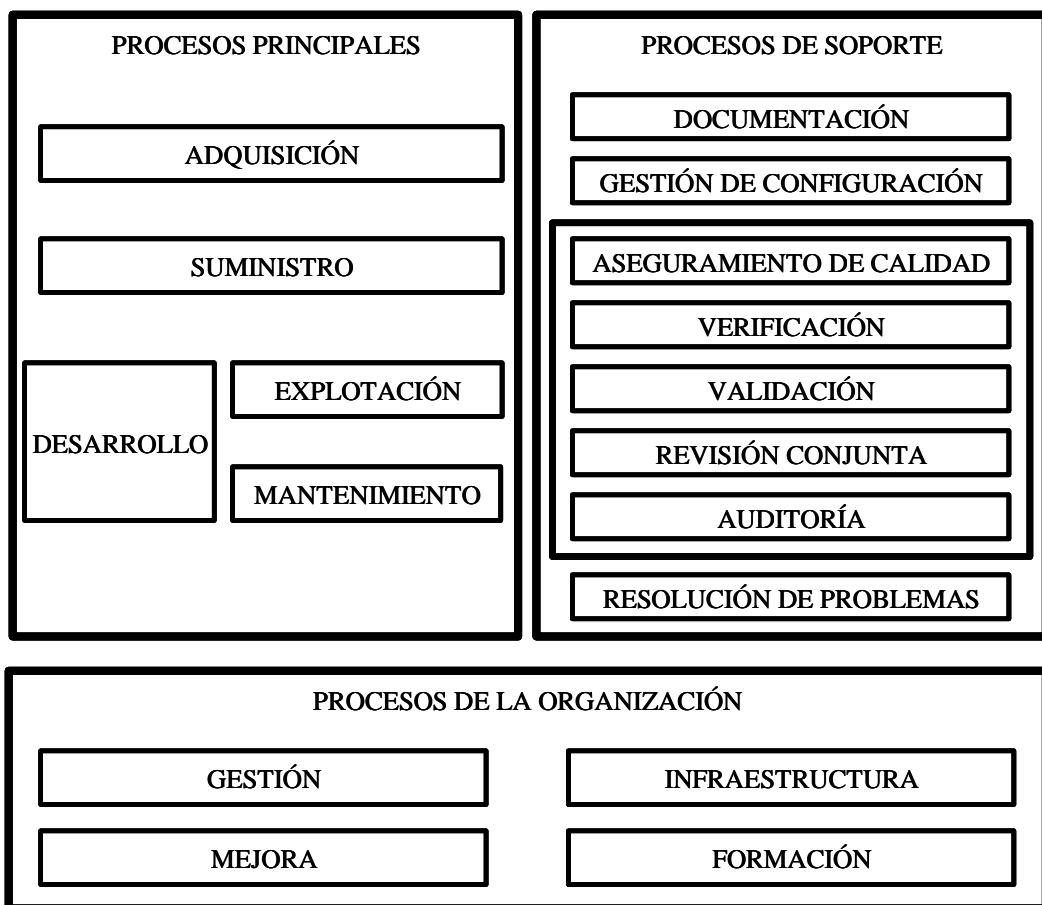


Figura 2.2.- Agrupación de procesos de la norma ISO 12207 - 1

2.2.2.1.- Procesos principales

Los procesos principales son aquellos que resultan útiles a las personas que inician o realizan el desarrollo, la explotación o el mantenimiento del software durante su ciclo de vida. Estas personas son los compradores, los suministradores, el personal de desarrollo, los usuarios y el personal de mantenimiento del software. Los procesos principales son:

2.2.2.1.1.- Proceso de adquisición

Este proceso contiene las actividades y las tareas que el comprador, el cliente o el usuario

realizan para adquirir un sistema o un producto software. También incluye la preparación y publicación de una solicitud de ofertas, la selección del suministrador del software y la gestión de los procesos desde la adquisición hasta la aceptación del producto.

2.2.2.1.2.- Proceso de suministro

Este proceso contiene las actividades y tareas que el suministrador realiza. Se inicia con la decisión de preparar una propuesta para responder a una petición de un comprador o por la firma de un contrato con el comprador para proporcionarle un producto software. También trata la identificación de los procedimientos y de los recursos necesarios para gestionar y garantizar el éxito del proyecto, incluyendo el desarrollo de los planes del proyecto y la ejecución de dichos planes hasta la entrega del producto software al comprador.

2.2.2.1.3.- Proceso de desarrollo

Este proceso contiene las actividades de análisis de requisitos, diseño, codificación, integración, pruebas e instalación y aceptación. Debido al interés que tiene este proceso, se resumen a continuación sus principales actividades:

- *Análisis de requisitos del sistema*: se especifican los requisitos del sistema, incluyendo las funciones y las capacidades que debe incluir, los requisitos de seguridad, de interacción hombre-máquina, de interfaces, de operaciones y de mantenimiento, las restricciones aplicables al diseño y los requisitos para su aceptación.
- *Diseño de la arquitectura del sistema*: se establece la arquitectura de alto nivel, la cual identificará los principales componentes de hardware y de software, así como las operaciones manuales del sistema.
- *Análisis de los requisitos del software*: se establecen y se documentan dichos requisitos, incluyendo la especificación de las características de calidad que debe cumplir el sistema, tales como: especificaciones funcionales y de la capacidad (por ejemplo, el rendimiento del sistema o aplicación, sus características físicas, etc.), interfaces externas, requisitos de aceptación, especificaciones de seguridad y protección (de la información, de daños personales, etc.), especificaciones de interacción hombre-máquina (tales como, operaciones manuales, restricciones personales, por ejemplo «sólo los directores podrán acceder a la aplicación X», etc.), definición de los datos que se van a manejar y requisitos de bases de datos (por ejemplo, la aplicación se hará con un gestor de bases de datos relacional), requisitos de instalación y de aceptación del software entregado (por ejemplo, la aplicación se ha de instalar en un entorno PC, y sólo se aceptará si el tiempo de respuesta para la operación X es menor de dos segundos), requisitos de explotación y de ejecución, y requisitos de mantenimiento del usuario.
- *Diseño de la arquitectura del software*: el desarrollador debe transformar los requisitos del software en una arquitectura o estructura de alto nivel que identifica sus componentes principales. También se elabora una versión preliminar de los manuales de usuario, se definen y documentan los requisitos que deben cumplir las pruebas de estos componentes y se planifica la integración del software.
- *Diseño detallado del software*: se realiza un diseño detallado para cada componente software, incluidas las bases de datos. Así mismo se actualizan los manuales de usuario, se definen y documentan los requisitos que deben cumplir las pruebas de estos componentes detallados y se planifican las pruebas unitarias del software.
- *Codificación y prueba del software*: se desarrollan y se documentan los distintos componentes software y las bases de datos. Posteriormente se prueba cada uno de los mismos para asegurar que satisfacen los requisitos. Además, se actualizan los manuales de

usuario.

- Integración del software: se integran los componentes del software y se prueban según sea necesario. También se actualizan los manuales de usuario.
- Prueba del software: el desarrollador lleva a cabo la prueba de cualificación en función de los requisitos especificados.
- Integración del sistema: se integran los elementos software y hardware junto con las operaciones manuales.
- Prueba del sistema: análoga a la del software, pero se llevará a cabo de acuerdo con los requisitos de cualificación especificados para el sistema (el software integrado con el resto de componentes del mismo: hardware, componentes físicos, etc.).
- Instalación del software en el entorno de explotación final donde vaya a funcionar. Cuando el software nuevo reemplace a un sistema existente, es recomendable mantener ambos sistemas en funcionamiento paralelo durante un período razonable de tiempo para comprobar el funcionamiento correcto del nuevo sistema.
- Soporte del proceso de aceptación del software: el desarrollador debe dar su apoyo a la revisión de aceptación y a la prueba del software por parte del comprador.

2.2.2.1.4.- Proceso de explotación

Este proceso incluye la explotación del software y el soporte operativo a los usuarios. Debido a que esta explotación está integrada en la del sistema, las actividades y tareas de este proceso se aplican al sistema completo. También se denomina proceso de operación.

2.2.2.1.5.- Proceso de mantenimiento

Este proceso aparece cuando el software necesita modificaciones, ya sea en el código o en la documentación asociada, debido a un error, una deficiencia, un problema o la necesidad de mejora o adaptación. El objetivo es modificar el software existente manteniendo su consistencia. Este proceso puede incluir también las actividades de migración a un nuevo entorno operativo y las de retirada del software.

2.2.2.2.- Procesos de soporte

Estos procesos sirven de apoyo al resto y se aplican en cualquier punto del ciclo de vida. Los procesos de soporte son los siguientes:

2.2.2.2.1.- Proceso de documentación

Este proceso registra la información producida por un proceso o actividad del ciclo de vida. Incluye todo el conjunto de actividades que permiten planificar, diseñar, desarrollar, producir, editar, distribuir y mantener los documentos necesarios para todas las personas involucradas: directores, ingenieros, personal de desarrollo, usuarios del sistema o software, etc.

2.2.2.2.2.- Proceso de gestión de la configuración

Este proceso aplica ciertos procedimientos administrativos y técnicos durante todo el ciclo de vida del sistema para:

- Identificar, definir y establecer la línea base de los elementos de configuración del software de un sistema.
- Controlar las modificaciones (por ejemplo, la modificación X al programa Y fue hecha por la persona Z) y las versiones de los elementos (por ejemplo, la última versión del programa X es la 1.4).

- Registrar e informar sobre el estado de los elementos y las peticiones de modificación.
- Asegurar la compleción, la consistencia y la corrección de los elementos.
- Controlar el almacenamiento, la manipulación y la entrega de los elementos.

2.2.2.2.3.- Proceso de aseguramiento de la calidad

Este proceso aporta una confianza en la que los procesos y los productos software del ciclo de vida cumplen con los requisitos especificados y se ajustan a los planes establecidos. El aseguramiento de la calidad puede ser interno o externo dependiendo de si la calidad del producto o proceso debe demostrarse a los directivos del proveedor o del comprador. El aseguramiento de la calidad puede utilizar los resultados de otros procesos de apoyo, como la verificación, la validación, las revisiones conjuntas, las auditorias y las actividades de resolución de problemas.

2.2.2.2.4.- Proceso de verificación

Este proceso determina si los requisitos de un sistema o del software están completos y son correcto, y si los productos software de cada fase del ciclo de vida cumplen los requisitos o condiciones impuestos sobre ellos en las fases previas (por ejemplo, si el código es coherente con el diseño). Este proceso puede ser responsabilidad de una organización de servicios, en cuyo caso, se denomina proceso de verificación independiente.

2.2.2.2.5.- Proceso de validación

Este proceso sirve para determinar si el sistema o software final cumplen con los requisitos previstos para su uso. Al igual que el anterior, este proceso puede ser ejecutado por una organización de servicios, en cuyo caso se denomina proceso de validación independiente.

2.2.2.2.6.- Proceso de revisión conjunta

Este proceso sirve para evaluar el estado del software y sus productos en una actividad del ciclo de vida o una fase de un proyecto. Las revisiones conjuntas se celebran tanto a nivel de gestión como a nivel técnico del proyecto a lo largo de todo su ciclo de vida.

2.2.2.2.7.- Proceso de auditoria

Este proceso permite determinar, en los hitos predeterminados, si se han cumplido los requisitos, los planes y el contrato.

2.2.2.2.8.- Proceso de resolución de problemas

Este proceso permite analizar y eliminar los problemas (disconformidades con los requisitos o el contrato) descubiertos durante el desarrollo, la explotación, el mantenimiento u otro proceso. El objetivo es aportar un medio oportuno y documentado para asegurar que todos los problemas descubiertos se analizan y solucionan.

2.2.2.3.- Procesos de la organización

Estos procesos los emplea una organización para llevar a cabo funciones tales como la gestión, la formación del personal o la mejora del proceso. Ayudan a establecer, implementar y mejorar la organización consiguiendo una organización más efectiva. Se llevan a cabo normalmente a nivel organizativo, fuera del ámbito de proyectos y contratos específicos.

2.2.2.3.1.- Proceso de gestión

Este proceso contiene las actividades y las tareas genéricas que puede emplear cualquier organización que tenga que gestionar su(s) proceso(s), por lo que incluye actividades como la planificación, el seguimiento y control, y la revisión y evaluación.

2.2.2.3.2.- Proceso de infraestructura

Este proceso establece la infraestructura necesaria para cualquier otro proceso, lo que incluye hardware, software, herramientas, técnicas, normas e instalaciones para el desarrollo, la explotación o el mantenimiento.

2.2.2.3.3.- Proceso de mejora

Este proceso sirve para establecer, valorar, medir, controlar y mejorar los procesos del ciclo de vida del software.

2.2.2.3.4.- Proceso de formación

Este proceso sirve para proporcionar y mantener al personal formado, lo que incluye el desarrollo del material de formación, así como la implementación del plan de formación.

2.2.3.- La norma ISO/IEC TR 15504-2 [Piattini 2003]

Los procesos de esta norma están fuertemente alineados con la anterior y de hecho ésta se puede considerar una extensión de la anterior en el sentido de que incorpora todo lo dicho para la anterior y añade o amplia procesos. Esta norma tiene además un carácter bidimensional ya que además de definir los procesos también expresa la capacidad que una empresa ha logrado en el desarrollo de dicho proceso. Esta dimensión será analizada con mayor nivel de detalle cuando se trate el CMM y el CMMI.

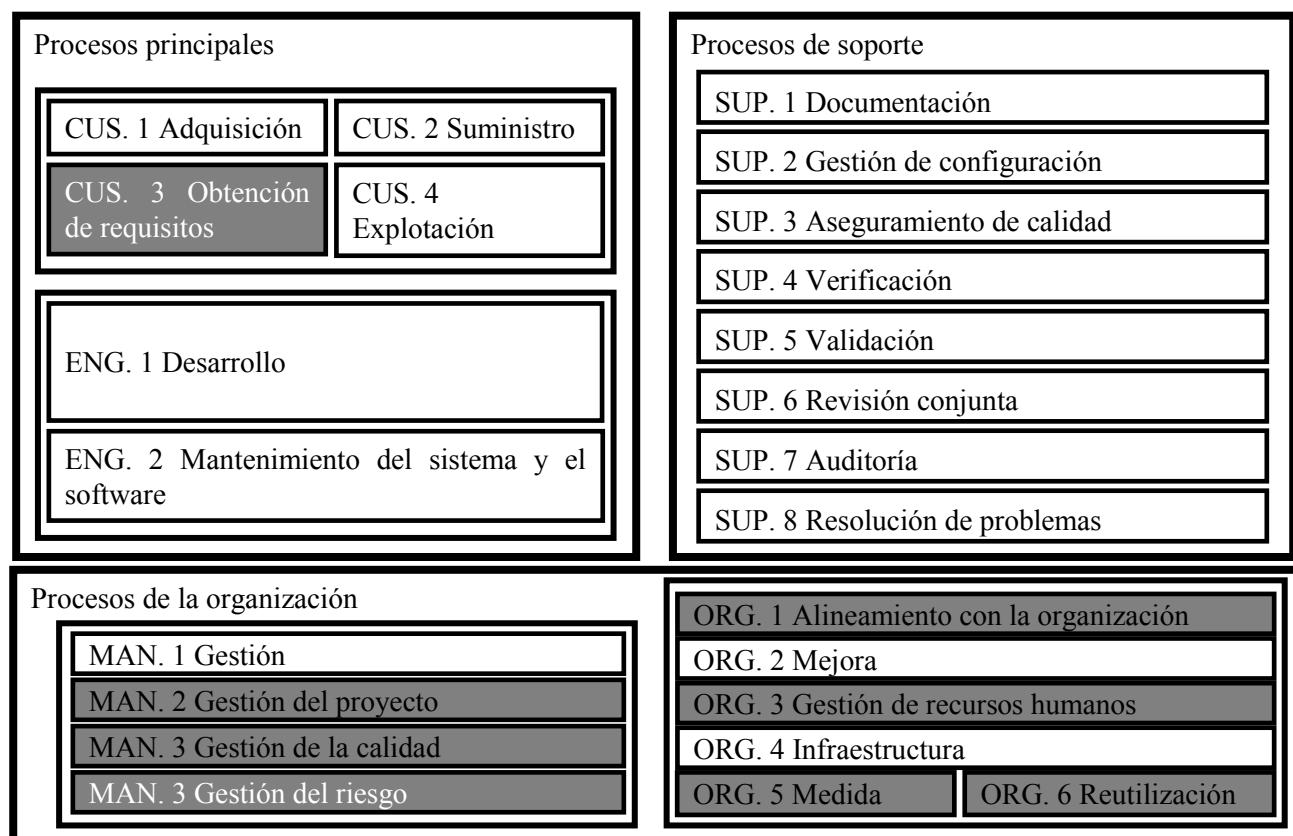


Figura 2.3.- Procesos de la ISO/IEC 15504-2 y su alineamiento con la ISO 12207-1

Los procesos que esta norma añade con respecto a la anterior son:

El proceso de obtención de requisitos, que se añade a los principales y tiene como objetivo reunir, procesar y seguir la evolución de las necesidades y requisitos del cliente a lo largo

de toda la vida del producto software y/o servicio, así como establecer una línea de base de requisitos que permita definir los entregables software requeridos.

Proceso de Alineamiento de la Organización. Se añada a los procesos de la organización y su propósito es asegurar que los individuos en la organización comparten una visión, cultura y comprensión común de los objetivos del negocio para autorizarlos a funcionar de forma efectiva. Aunque la reingeniería del negocio y la Gestión de la Calidad Total tienen un alcance mucho mayor que el de proceso software, la mejora de proceso software ocurre en un contexto de negocio y, para tener éxito, debe tratar los objetivos del negocio.

Proceso de Gestión de Recursos Humanos. Se añada a los procesos de la organización y su propósito es proporcionar a la organización y a los proyectos con individuos que posean las habilidades y el conocimiento para realizar sus roles de forma efectiva y para trabajar juntos como un grupo.

Proceso de Medida. Se añada a los procesos de la organización y su propósito es recoger y analizar los datos relativos a los productos desarrollados y a los procesos implementados dentro de la unidad organizativa, para soportar la gestión efectiva de los procesos y para demostrar de forma objetiva la calidad de los productos.

Proceso de Reutilización. Se añada a los procesos de la organización y su propósito es promover y facilitar la reutilización de productos de trabajo software nuevos y existentes desde una perspectiva de la organización y del producto/proyecto.

El proceso de gestión de los procesos de la organización se descompone ahora en:

Proceso de Gestión (proceso básico). El propósito del Proceso de Gestión es organizar y controlar la iniciación y la realización de cualquier proceso o función dentro de la organización para lograr sus objetivos y los objetivos de negocio de la organización de manera efectiva.

Proceso de Gestión del Proyecto (proceso nuevo). El propósito del Proceso de Gestión del Proyecto (véase el Capítulo 5) es identificar, establecer, coordinar y controlar las actividades, tareas y recursos necesarios para que un proyecto produzca un producto y/o servicio que cumplan los requisitos.

Proceso de Gestión de la Calidad (proceso nuevo). El propósito del Proceso de Gestión de la Calidad (véase el Capítulo 12) es controlar la calidad de los productos y/o servicios del proyecto y asegurar que satisfacen al cliente. El proceso implica el establecimiento de un enfoque sobre el control de la calidad del producto y del proceso tanto a nivel del proyecto como a nivel de la organización.

Proceso de Gestión del Riesgo (proceso nuevo). El propósito del Proceso de Gestión del Riesgo es identificar y reducir continuamente los riesgos en un proyecto a lo largo de su ciclo de vida. El proceso implica el establecimiento de un enfoque en el control de los riesgos tanto a nivel del proyecto como a nivel de la organización.

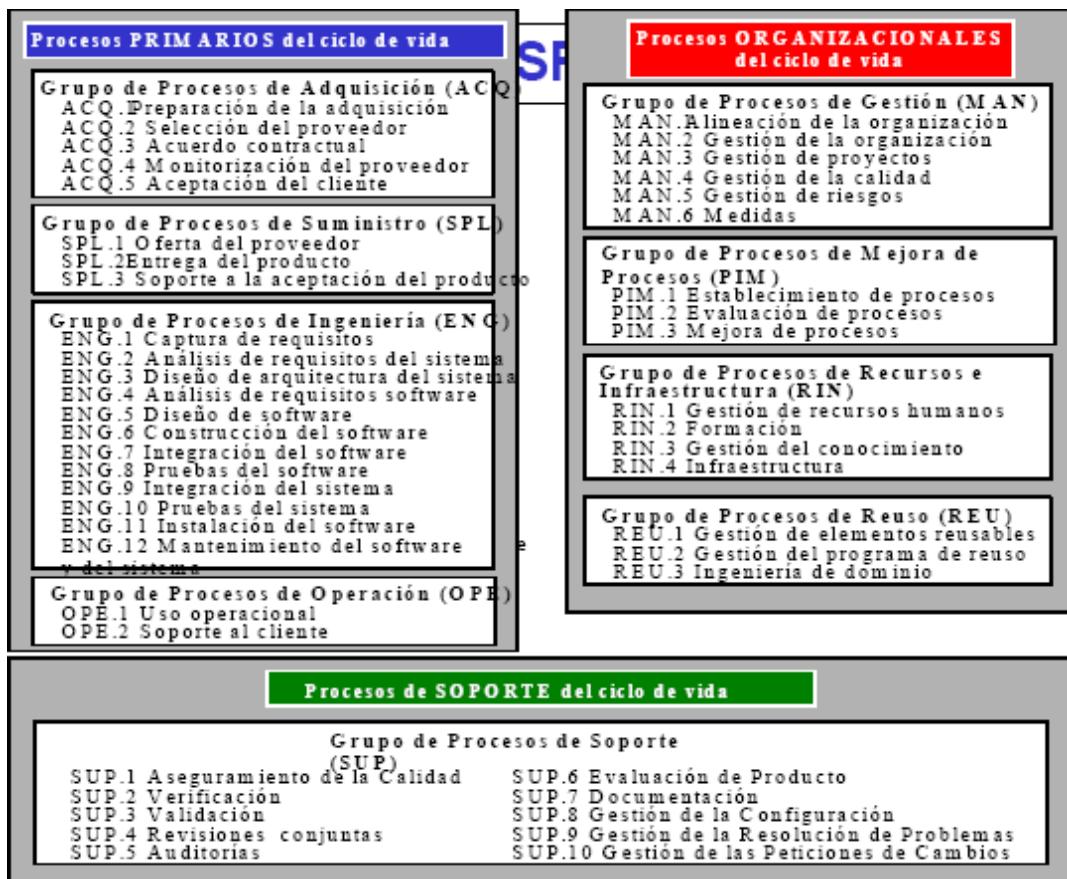


Figura 2.4. Procesos de la norma ISO/IEC 15504-2 (rev. 2003)

2.3.- Evaluación del proceso software.

La crisis del software y los problemas generados por del software heredado, planteados en temas anteriores, dan cuenta detallada de los problemas generados por desarrollar software sin seguir un proceso o una metodología orientados a la calidad. El desarrollo de software ad hoc supone un riesgo importante para las empresas que no pueden conocer el progreso de sus proyectos ni predecir su calidad, que en general será baja. Como consecuencia los presupuestos del proyecto no se cumplen, los requisitos o las restricciones del sistema no se alcanzan y el proyecto fracasa con el efecto inmediato de la insatisfacción del cliente.

De todo lo anterior se dedujo un interés en las empresas por encontrar y seguir algún proceso que les garantizara los resultados. A pesar de este interés la aplicación de los procesos y su adaptación a las empresas y sus políticas no resulta trivial. En muchos casos las empresas conocen los modelos y reconoces su utilidad pero los aplican de forma incompleta e inconsistente según sus necesidades más acuciantes. Como consecuencia en muchos casos las empresas llegan a desconocer en qué punto se encuentran sus procesos de forma que sus objetivos no se centran en el proceso sino en saber si lo siguen y adaptarlo de forma consistente a sus propias estrategias.

Surgen de este modo una serie de modelos orientados a guiar a las empresas en su evolución desde su estado actual, que debe ayudar a determinar el modelo, hasta uno evolucionado que garantice la optimización en la producción de software de calidad. El modelo guiará a la empresa en la mejora del proceso y, según su evolución, hasta le permitirá decidir el camino que prefiera seguir. La relación entre los procesos y los métodos de evaluación se muestra en la siguiente figura.

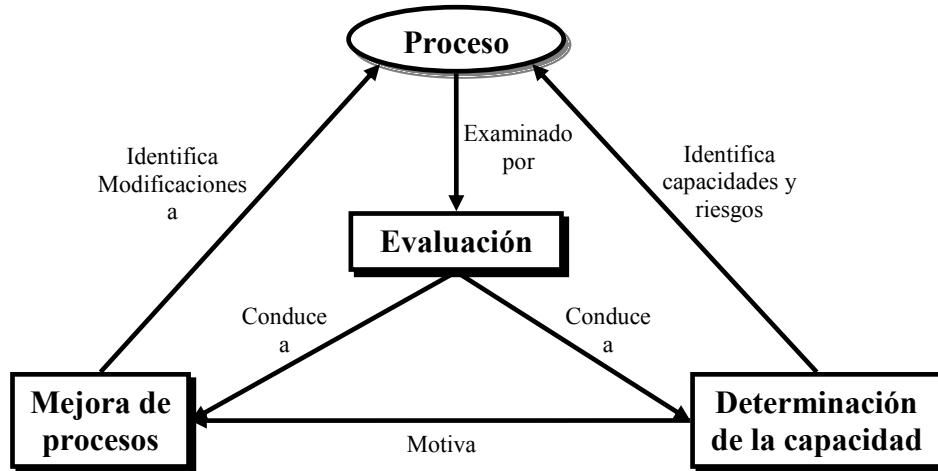


Figura 2.5. Relación con los procesos de evaluación

2.3.1.- Integración del Modelo de Capacidad de Madurez (CMMI)

El modelo CMM ha evolucionado actualmente al CMMI (Capability Maturity Model Integration) [Pressman06]. Este modelo presenta en contra de lo que sucedía con su primera versión dos formas diferentes: un modelo continuo y un modelo discreto. En el modelo continuo se evalúa las distintas áreas del proceso en 6 niveles de capacidad. El CMMI v1.2 propone 22 áreas del proceso (PA) divididas en 4 categorías (ver tabla 2.6) cada una de las cuales puede estar en cualquiera de los 6 niveles de capacidad mencionados y que detallamos a continuación:

Nivel 0: Incompleto. El área del proceso (por ejemplo, la gestión de requisitos) aún no se realiza o todavía no alcanza todas las metas y objetivos definidos para el nivel 1 de capacidad.

Nivel 1: Realizado. Todas las metas específicas del área del proceso (como las definió el CMMI) han sido satisfechas. Las tareas de trabajo requeridas para producir el producto específico han sido realizadas.

Nivel 2: Administrado. Todos los criterios del nivel 1 han sido satisfechos. Además, todo el trabajo asociado con el área de proceso se ajusta a una política organizacional definida; toda la gente que ejecuta el trabajo tiene acceso a recursos adecuados para realizar su labor; los clientes están implicados de manera activa en el área de proceso, cuando esto se requiere; todas las tareas de trabajo y productos están "monitorizados, controlados y revisados; y son evaluados de acuerdo a la descripción del proceso" [CMM02].

Nivel 3: Definido. Todos los criterios del nivel 2 se han cumplido. Además, el proceso está "adaptado al conjunto de procesos estándar de la organización, de acuerdo con las políticas de adaptación de esta misma, y contribuye a la información de los productos del trabajo, mediciones y otras mejoras del proceso para los activos del proceso organizacional" [CMM02].

Nivel 4: Administrado en forma cuantitativa. Todos los criterios del nivel 3 han sido cumplidos. Además, el área del proceso se controla y mejora mediante mediciones y evaluación cuantitativa. "Los objetivos cuantitativos para la calidad y el desempeño del proceso están establecidos y se utilizan como un criterio para administrar el proceso" [CMM02].

Nivel 5: Mejorado. Todos los criterios del nivel 4 han sido satisfechos. Además, el área del proceso "se adapta y mejora mediante el uso de medios cuantitativos (estadísticos) para conocer las necesidades cambiantes del cliente y mejorar de manera continua la eficacia del área del proceso que se está considerando" [CMM02].

Categoría 1	The Process Management process areas of CMMI are as follows
área 1	Organizational Process Focus
área 2	Organizational Process Definition
área 3	Organizational Training
área 4	Organizational Process Performance
área 5	Organizational Innovation and Deployment
Categoría 2	The Project Management process areas of CMMI are as follows
área 6	Project Planning
área 7	Project Monitoring and Control
área 8	Supplier Agreement Management
área 9	Integrated Project Management + IPPD (or Integrated Project Management)
área 10	Risk Management
área 11	Quantitative Project Management
Categoría 3	The Engineering process areas of CMMI are as follows
área 12	Requirements Development
área 13	Requirements Management
área 14	Technical Solution
área 15	Product Integration
área 16	Verification
área 17	Validation
Categoría 4	The Support process areas of CMMI are as follows
área 18	Configuration Management
área 19	Process and Product Quality Assurance
área 20	Measurement and Analysis
área 21	Decision Analysis and Resolution
área 22	Causal Analysis and Resolution

Figura 2.6. Áreas de proceso por categoría para el CMMI continuo.

El CMMI define cada área del proceso en función de "metas específicas" (ME) y de las "prácticas específicas" (PE) requeridas para alcanzar dichas metas. Las metas específicas establecen las características que deben existir para que las actividades implicadas por un área de proceso sean efectivas. Las prácticas específicas convierten una meta en un conjunto de actividades relacionadas con el proceso.

Por ejemplo, para la planificación del proyecto (en la tabla PA 6) las metas específicas (ME) y sus prácticas específicas asociadas (PE) que se han definido son [CMM02]:

ME 1 Establecer estimaciones

- PE 1.1 -1 Estimar el alcance del proyecto.
- PE 1.2-1 Establecer estimaciones para los atributos del producto y las tareas del trabajo.
- PE 1.3- 1 Definir el ciclo de vida del proyecto.
- PE 1.4- 1 Determinar estimaciones de esfuerzo y costo.

ME 2 Desarrollar un plan de proyecto

- PE 2.1- 1 Establecer el presupuesto y el programa.
- PE 2.2- 1 Identificar los riesgos del proyecto.
- PE 2.3- 1 Planear la gestión de los datos.
- PE 2.4- 1 Planear los recursos del proyecto.
- PE 2.5- 1 Planear los conocimientos y habilidades que se requieren.
- PE 2.6- 1 Planear la participación del cliente.
- PE 2.7-1 Establecer el plan de proyecto.

ME 3 Comprometerse con la planificación.

- PE 3.1-1 Revisar los planes que afectan al proyecto.
- PE 3.2-1 Conciliar el trabajo y los niveles de recursos.
- PE 3.3-1 Comprometerse con la planificación.

Además de las metas y prácticas específicas, el CMMI también define una serie de cinco

metas genéricas (MG) y prácticas genéricas (PG) relacionadas con cada área del proceso. Cada una de las metas genéricas corresponde a uno de los cinco niveles de capacidad. Por lo tanto, para lograr un nivel de capacidad particular se debe alcanzar la meta genérica para ese nivel y las prácticas genéricas que corresponden a esa meta. Para ilustrar lo anterior, a continuación se enumeran las metas genéricas (MG) y las prácticas genéricas (PG) para el área del proceso de planeación del proyecto [CMM02]:

MG 1 Alcanzar las metas específicas

PG 1.1 Realizar prácticas base.

MG 2 institucionalizar un proceso de gestión

PG 2.1 Establecer una política organizacional.

PG 2.2 Planejar el proceso.

PG 2.3 Proporcionar recursos.

PG 2.4 Asignar responsabilidades.

PG 2.5 Capacitar gente.

PG 2.6 Manejar configuraciones.

PG 2.7 Identificar y hacer participar a clientes.

PG 2.8 Monitorizar y controlar el proceso.

PG 2.9 Evaluar la adherencia de un modo objetivo.

PG 2.10 Revisar el estatus con un alto grado de gestión.

MG 3 Institucionalizar un proceso definido

PG 3.1 Establecer un proceso definido.

PG 3.2 Recolectar información de la mejoría.

MG 4 Institucionalizar un proceso manejado en forma cuantitativa

PG 4.1 Establecer objetivos cuantitativos para el proceso.

PG 4.2 Estabilizar el desempeño del subproceso.

MG 5 Institucionalizar un proceso de mejoramiento.

PG 5.1 Asegurar la mejora continua del proceso.

PG 5.2 Corregir las causas de los problemas desde la raíz

El modelo discreto del CMMI define las mismas áreas, metas y prácticas específicas del proceso que el modelo continuo. En él, sin embargo, sólo están definidas las Metas Generales 2 y 3 tal y como se han descrito para el modelo continuo. La principal diferencia es que el modelo discreto establece cinco niveles de madurez, en vez de cinco niveles de capacidad. Para lograr un nivel de madurez se deben conseguir metas y prácticas específicas relacionadas con un conjunto de áreas del proceso. Para el nivel de madurez 2 se deben cumplir además las metas generales 2 para cada una de las áreas de proceso de ese nivel y para el nivel 3 y subsiguientes se deben satisfacer también las metas generales 3 para cada área de proceso presente en el nivel. De esta forma el nivel de madurez 2 implica realizar las metas generales 2 para 7 áreas de proceso mientras que el nivel 3 implica cumplir las metas generales 2 y 3 para 18 áreas de proceso. Los niveles 4 y 5 sólo añaden respectivamente 2 áreas de proceso cada uno para las que, por supuesto, se deben cumplir las metas generales 2 y 3.

La relación de las áreas del proceso presentes en cada nivel se muestra en la figura 2.7

Nivel	Enfoque	Áreas del proceso
De optimización	Mejora continua del proceso	Innovación organizacional y despliegue. Análisis causal y resolución
Gestionado de modo cuantitativo	Gestión cuantitativa	Ejecución del proceso organizacional. Gestión cuantitativa del proyecto
Definido	Estandarización del proceso	Desarrollo de requisitos Solución técnica Integración del producto Verificación

		Validación Enfoque del proceso organizacional Definición del proceso organizacional Capacitación organizacional Gestión integrada del proyecto Gestión integrada del proveedor Gestión del riesgo Análisis y resolución de la decisión Ambiente organizacional para la integración Equipo integrado
Gestionado	Gestión básica del proyecto	Gestión de requisitos Planificación del proyecto Monitorización y control del proyecto Gestión de acuerdos del proveedor Medición y análisis Aseguramiento de la calidad del producto y del proceso Gestión de la configuración
Ejecutado		

Figura 2.7. PAs por nivel de madurez del modelo CMMI discreto.

Tema 3. Ingeniería de software. Ciclos de vida

Tema 3. Ingeniería de software. Ciclos de vida	i
3.1.- Modelos del ciclo de vida del software.....	33
3.1.1.- Paradigma del ciclo de vida en cascada	33
3.1.2.- Paradigma de la construcción de prototipos.....	36
3.1.3.- Ciclo de vida incremental.....	40
3.1.4.- Uso de técnicas de cuarta generación.....	40
3.1.5.- El modelo en espiral.....	43
3.2.- Desarrollo ágil.....	48
3.2.1.- Modelado Ágil.....	52
3.2.2.- Programación extrema.....	53

Figura 3.1. Ciclo de vida en cascada.....	33
Figura 3.2. Filosofía de desarrollo del paradigma de prototipos.....	37
Figura 3.3. Construcción de prototipos.....	38
Figura 3.4. El modelo incremental.....	40
Figura 3.5. Ciclo de vida usando técnicas de cuarta generación.....	41
Figura 3.6. El modelo en espiral.....	44
Figura 3.7. Ciclo de la programación extrema.....	53

3.1.- 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.

3.1.1.- 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.

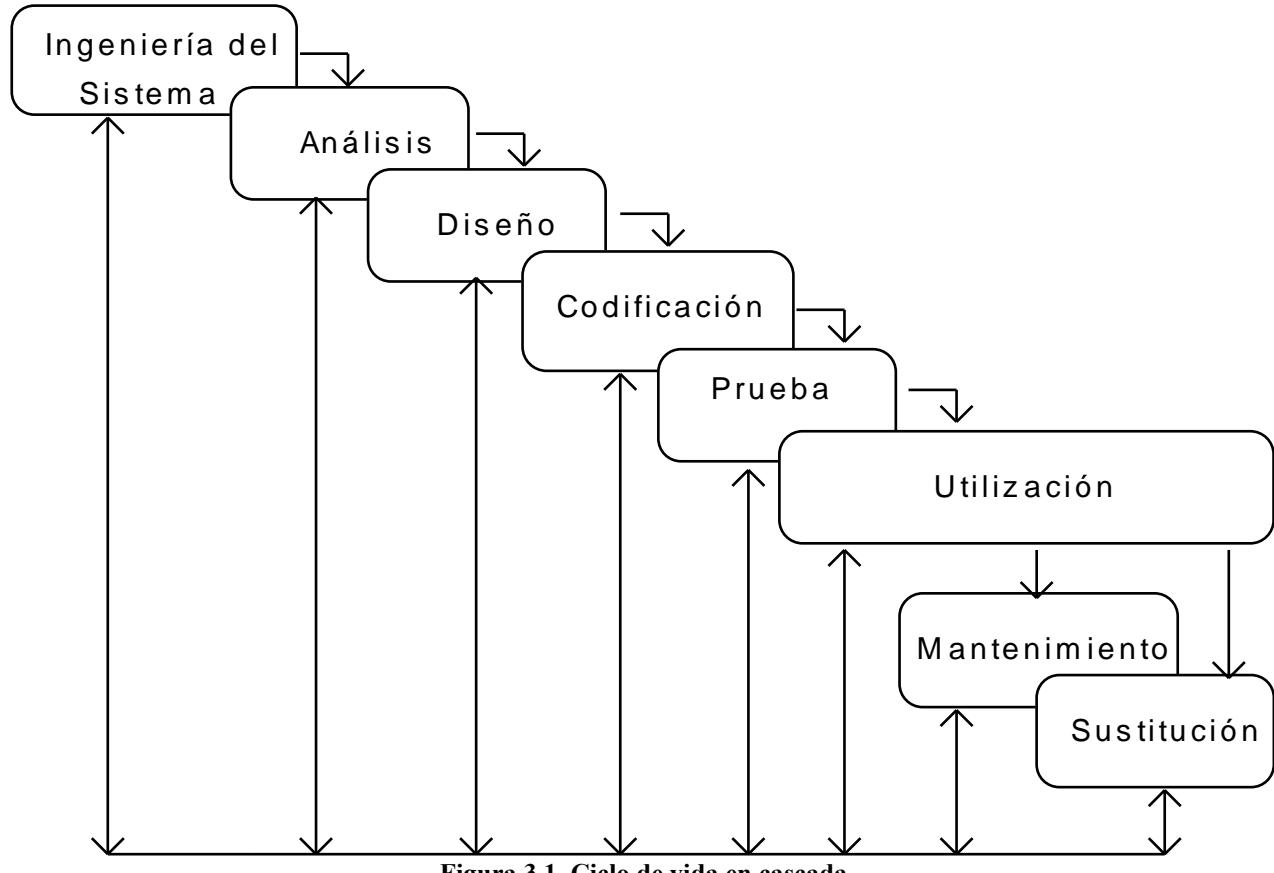


Figura 3.1. Ciclo de vida en cascada.

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:

3.1.1.1.- 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.

3.1.1.2.- 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.

3.1.1.3.- 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).

3.1.1.4.- 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.

3.1.1.5.- 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.

3.1.1.6.- 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.

3.1.1.7.- 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

3.1.1.8.- Sustitución.

La vida del software no es ilimitada y cualquier aplicación, por buena que sea, acaba por ser sustituida por otra más amplia, más rápida o más bonita y fácil de usar.

La sustitución de un software que está funcionando por otro que acaba de ser desarrollado es una tarea que hay que planificar cuidadosamente y que hay que llevar a cabo de forma organizada. Es conveniente realizarla por fases, si esto es posible, no sustituyendo todas las aplicaciones de golpe, puesto que la sustitución conlleva normalmente un aumento de trabajo para los usuarios, que

tienen que acostumbrarse a las nuevas aplicaciones, y también para los 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 más 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.

3.1.2.- 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 3.2. 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 y los listados del sistema.
- 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.

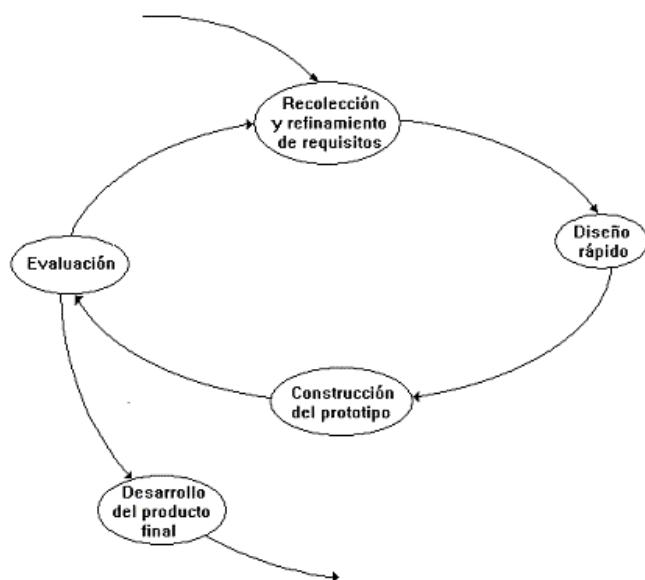


Figura 3.3. Construcción de prototipos

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 conocemos. 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 desechariendo 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 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.

3.1.3.- 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.

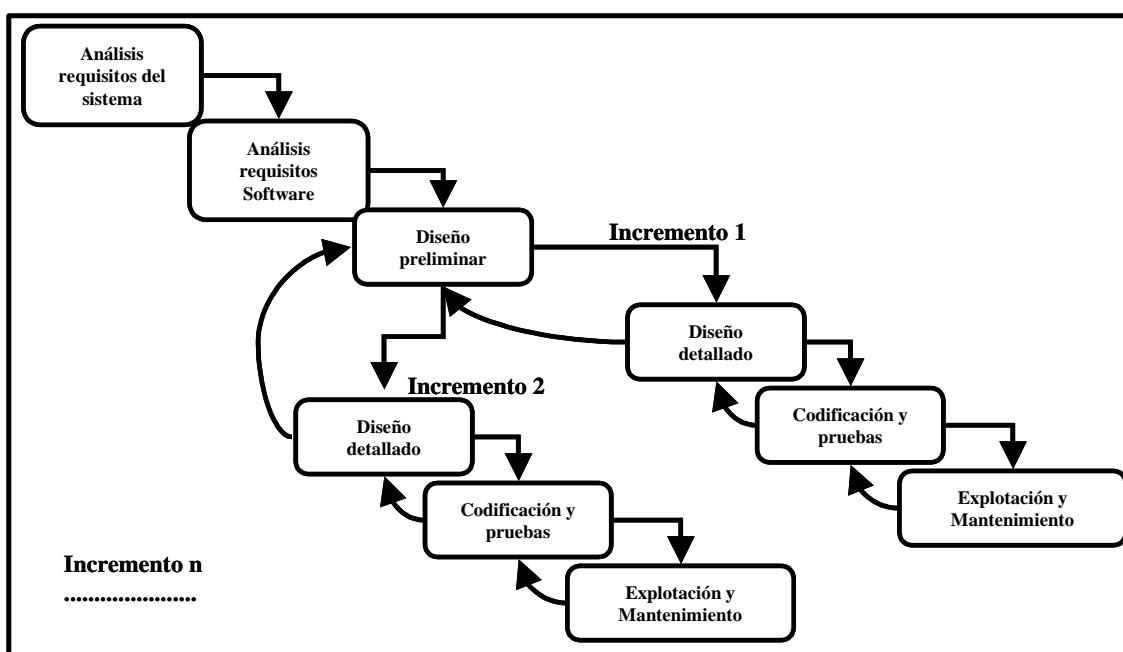


Figura 3.4. El modelo incremental.

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.

3.1.4.- 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.).

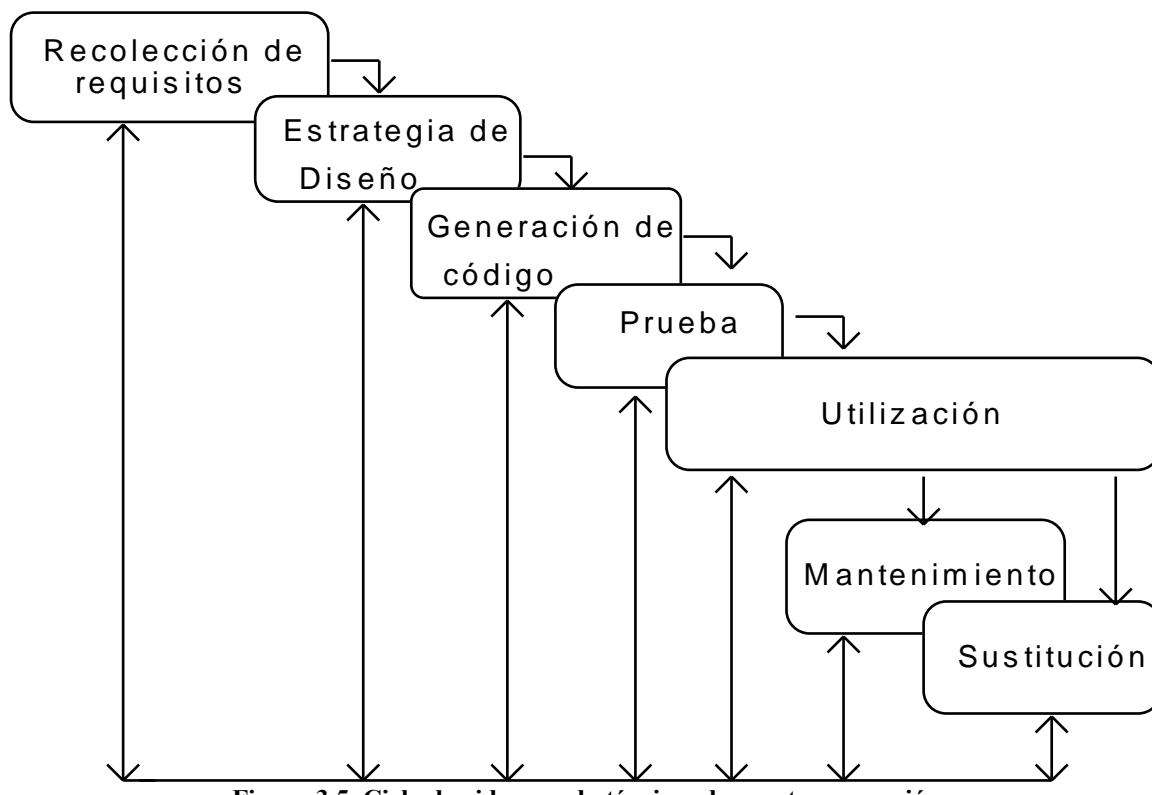


Figura 3.5. Ciclo de vida usando técnicas de cuarta generación.

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.

3.1.5.- 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.

3.1.5.1.- 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 explícitas 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.

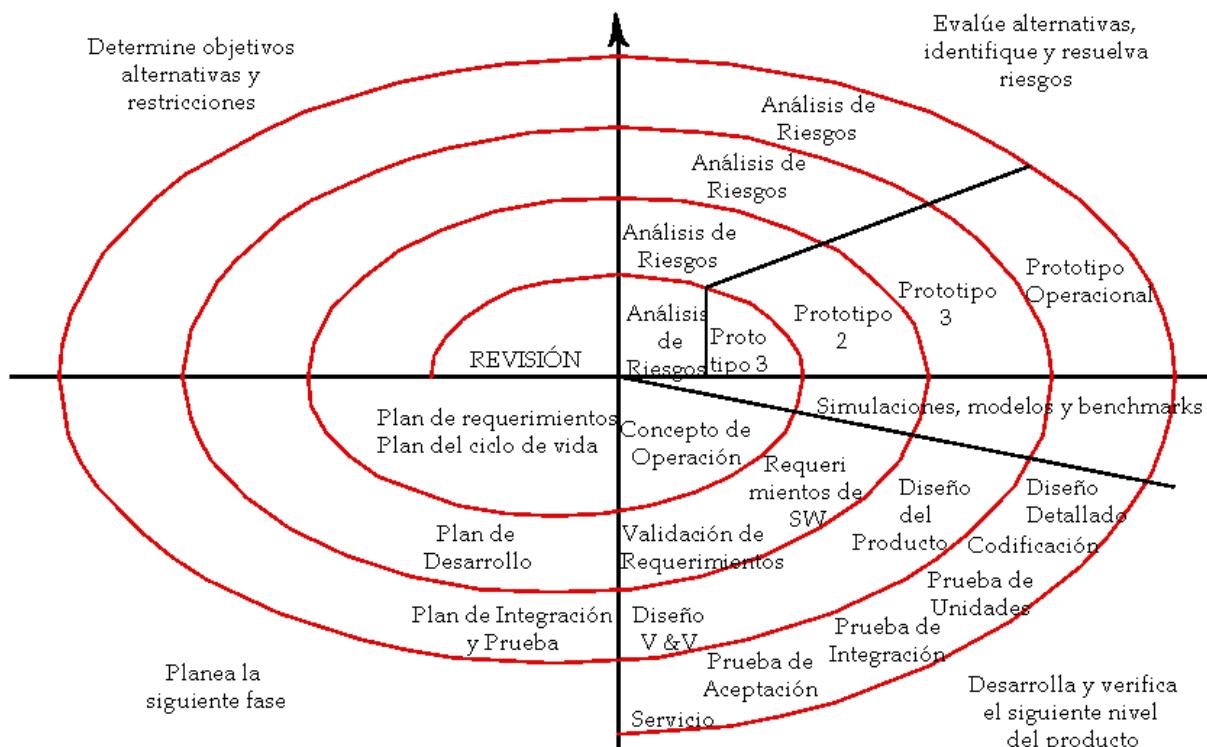


Figura 3.6. El modelo en espiral.

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 3.6, 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 de 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.

3.1.5.2.- 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 implementen. 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.

3.2.- Desarrollo ágil.

[Pressman 6^a, 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 immutable 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 cuánto 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] puntuiza 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.

3.2.1.- 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.

3.2.2.- 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.

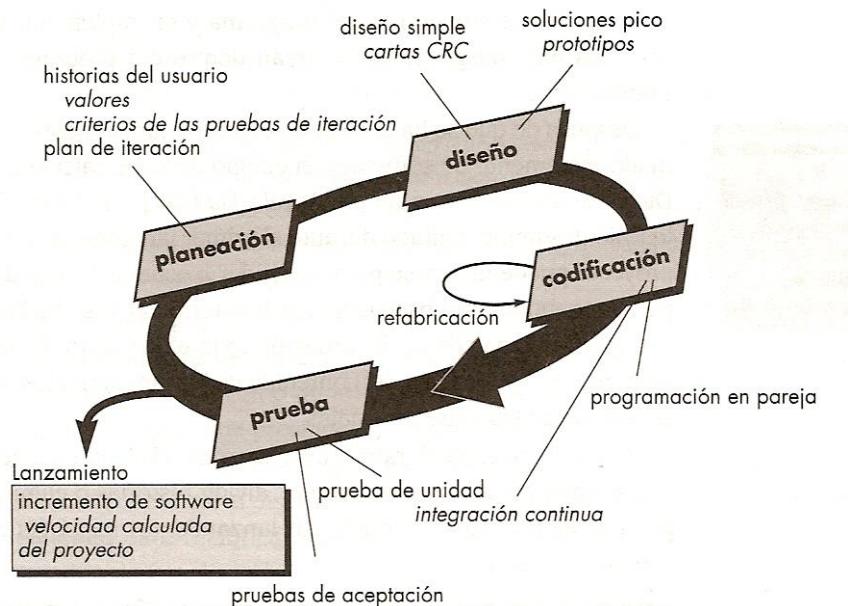


Figura 3.7. Ciclo de la programación extrema

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 qué 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 tema 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.

Tema 4.- Análisis de requisitos

4.1.- Introducción.....	65
4.2.- Primeras cuestiones	66
4.2.1.-Los requisitos.....	66
4.2.2.-La duración del análisis.	68
4.2.3.-El personal implicado.	69
4.2.4.-Dificultades del análisis de requisitos.	69
4.3.- Recogida inicial de requisitos.....	70
4.3.1.-Método de trabajo TFEA.....	72
4.4.- Análisis de requisitos del sistema.....	75
4.4.1.-Fases del análisis de sistemas.....	75
4.4.1.1.- FASE: Identificación de las necesidades.....	75
4.4.1.2.- FASE: Asignación de funciones.....	79
4.4.2.-Estudio de viabilidad.	80
4.4.2.1.- Análisis económico.	81
4.4.2.2.- Análisis técnico.	83
4.4.3.-Representación de la arquitectura del sistema.....	83
4.4.4.-Especificación del sistema.....	86
4.5.- Análisis de requisitos del software	88
4.5.1.-Objetivos y actividades del análisis de requisitos del software.....	88
4.5.2.-Principios del análisis de requisitos del software.....	90
4.6.- Especificación de requisitos.	92
4.6.1.-Principios de la especificación	93
4.6.2.-Características del documento de especificación.	95
4.6.3.-Estructura para la ERS.....	98
4.7.- Validación de requisitos	100
4.7.1.-Revisión de requisitos.....	101
4.8.- Administración de requisitos	102
4.8.1.-Requisitos duraderos y volátiles	102
4.8.2.-Planificación de la administración de requisitos.	103
4.8.3.-Administración del cambio de requisitos.	105

4.1.-Introducción

Según vimos en el tema 2 todas las normas identificaban uno o varios procesos relacionados con el Análisis de requisitos. Del mismo modo, en el tema 3, hemos presentado diversos modelos de ciclo de vida para el desarrollo de software y, en todos ellos, se puede observar la existencia de una fase denominada Análisis de requisitos.

Entre las tareas que hay que realizar en esta fase están el estudio de las características y la función del sistema, la definición de los requisitos del software y del sistema del que el software forma parte, así como la planificación inicial del proyecto y, posiblemente, algunas tareas relacionadas con el análisis de riesgos.

Todas estas tareas deben realizarse al comienzo del proyecto, pero el principal problema que se nos presenta es que, en estos momentos iniciales, es difícil tener una idea clara o, al menos, es difícil llegar a expresarla de cuáles son los requisitos del sistema y del software, y llegar a comprender en su totalidad la función que el software debe realizar. Por esto, algunos de los modelos de ciclo de vida estudiados proponen enfoques cíclicos de refinamiento de los requisitos o incluso de todo el proceso de desarrollo de software.

El análisis de requisitos es el primer paso técnico del proceso de ingeniería del software. Es aquí donde se refina la declaración general del ámbito del software en una *especificación* concreta que se convierte en la base de todas las actividades de ingeniería del software que siguen. Algunos modelos de ciclo de vida distinguen una fase de análisis de requisitos y otra de especificación del sistema. En cualquier caso, el análisis del sistema concluye con la especificación de los requisitos del mismo.

El análisis se centra en los ámbitos de información, funcional y de comportamiento del problema en cuestión. Para comprender mejor lo que se requiere se divide el problema en partes y se desarrollan representaciones o modelos que muestran la esencia de los requisitos (modelo esencial) y, posteriormente, los detalles de implementación (modelo de implementación). Como resultado del análisis se desarrolla la *especificación de requisitos*, un documento que describe el problema analizado y muestra la estructura de la solución propuesta.

La forma de especificar un sistema tiene una gran influencia en la calidad de la solución implementada finalmente. Tradicionalmente los ingenieros de software han venido trabajando con especificaciones incompletas, inconsistentes o erróneas, lo que invariablemente lleva a la confusión y a la frustración en todas las etapas del ciclo de vida. Como consecuencia de esto, la calidad, la corrección y la completitud del software disminuyen.

Las técnicas de análisis que veremos conducen a una especificación en papel o basada en ordenador (si utilizamos herramientas de análisis o CASE), que contiene descripciones gráficas y textuales (normalmente en lenguaje natural o alguna forma de pseudocódigo) de los requisitos del software. Por otro lado, la construcción de prototipos lleva a una especificación ejecutable, es decir, el prototipo sirve como representación de los requisitos. Por último, los lenguajes de especificación formal

conducen a representaciones de los requisitos que pueden ser analizadas o verificadas formalmente. Sin embargo, sea cual sea el método elegido, existen una serie de características comunes:

1. Realizan una abstracción de las características del sistema, es decir, consisten en desarrollar un modelo del mismo.
2. Representan el sistema de forma jerárquica, basándose en mecanismos de partición del problema y estableciendo varios niveles de detalle.
3. Definen cuidadosamente las interfaces del sistema, tanto las interfaces externas, que relacionan el sistema con su entorno, como de las internas, las que se establecen entre los distintos módulos definidos.
4. Sirven de base para las etapas posteriores de diseño y de implementación.
5. Plantean varios niveles de especificación. Requisitos de usuario, requisitos de sistema, esenciales y de implementación.
6. No prestan demasiada atención a la representación de las restricciones o de criterios de validación (exceptuando los métodos formales).

Es por esta última deficiencia por la que surge la necesidad de los métodos de especificación formal, especialmente en aquellos sistemas en los que la corrección, completitud o fiabilidad del software juegan un papel fundamental. Ejemplos de este tipo de sistemas pueden ser los protocolos de comunicación, el software de control de una central nuclear o el de gestión del tráfico aéreo.

4.2.- Primeras cuestiones

El análisis de requisitos constituye el primer intento de comprender cuál va a ser la función y ámbito de información de un nuevo proyecto. Indudablemente, los esfuerzos realizados en esta fase producen beneficios en las fases posteriores. Aunque el desarrollo de esta fase plantea muchas cuestiones empezaremos por lo más básico:

4.2.1.- Los requisitos.

Obviamente la primera pregunta que cabe hacerse es: ¿Qué es un requisito?. Éste se puede definir como: Una condición o capacidad que necesita el usuario para resolver un problema o conseguir un objetivo determinado. Por extensión las condiciones que debe cumplir o poseer un sistema o uno de sus componentes para satisfacer un contrato, una norma o una especificación (Piattini 2003).

En Sommersville encontramos una clasificación para los requisitos. En ella se distingue entre requisitos funcionales, no funcionales y requisitos de dominio:

- **Requisitos funcionales** Son declaraciones de los servicios que proveerá el sistema, de la manera en que éste reaccionará a entradas particulares y de cómo se comportará en situaciones particulares. En algunos casos, los requisitos funcionales de los sistemas también declaran explícitamente lo que el sistema no debe hacer.

- **Requisitos no funcionales** Son restricciones de los servicios o funciones ofrecidos por el sistema. Incluyen restricciones de tiempo, sobre el proceso de desarrollo, estándares, etcétera.
- **Requisitos del dominio** Son requisitos que provienen del dominio de aplicación del sistema y que reflejan las características de ese dominio. Estos pueden ser funcionales o no funcionales.

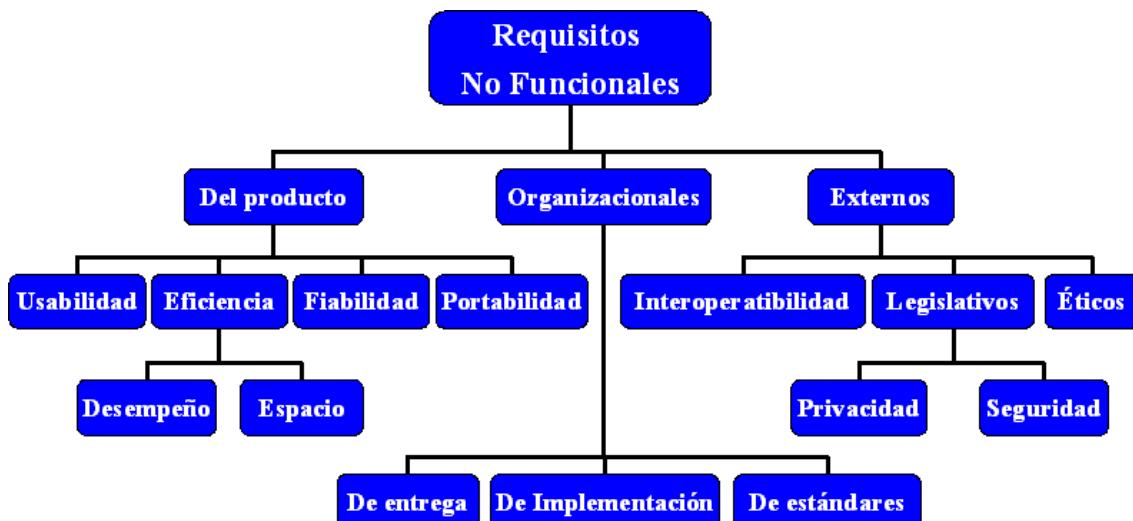
En realidad, la distinción entre estos tipos diferentes de requisitos no es tan clara como sugieren estas definiciones y con frecuencia al concretar requisitos de un tipo aparecen requisitos de otro.

Los requisitos funcionales de un sistema describen la funcionalidad o los servicios que se espera que éste provea. Estos, para un sistema de software, se expresan de diferentes formas. Cuando se expresan en lenguaje natural estos contienen con frecuencia ambigüedades e inconsistencias que son la fuente de muchos de los problemas de las ingenierías.

El sistema deberá proveer “visores adecuados” para que el usuario lea documentos en el almacén de documentos

Cuando un requisito funcional no se identifica o no se cumple el sistema no contiene toda su funcionalidad lo que supone una degradación del sistema que será tanto mayor cuantos más sean los requisitos incumplidos.

Los requisitos no funcionales son aquellos que no se refieren directamente a las funciones específicas que entrega el sistema, sino a las propiedades emergentes de éste como la fiabilidad, la respuesta en el tiempo y la capacidad de almacenamiento. Dicho de otra forma, definen las restricciones del sistema como la capacidad de los dispositivos de entrada/salida y la representación de los datos que se utiliza en las interfaces del sistema. Los requisitos no funcionales con frecuencia hacen referencia al sistema como un todo por lo que los fallos en ellos inutilizan el sistema lo que los convierte en más críticos que los requisitos funcionales.



La figura muestra una posible clasificación de los requisitos no funcionales en la que se aprecia claramente como estos no están únicamente relacionados con el producto

sino también con el entorno (organización y exterior) en el que se desarrolla y ejecuta la aplicación.

A menudo, los requisitos no funcionales entran en conflicto e interactúan con otros requisitos funcionales del sistema. En principio los requisitos funcionales y no funcionales se diferencian en el documento de requisitos. En la práctica, esto es difícil. Si se declaran de forma separada es difícil ver la relación entre ellos. Si se declaran en conjunto es difícil separar las consideraciones funcionales y no funcionales e identificar correctamente los requisitos. Se debe hallar un balance apropiado que permita resaltar claramente las propiedades emergentes del sistema, colocando, por ejemplo, estos requisitos en una sección aparte.

Finalmente los requisitos del dominio, son como su nombre indica, dependientes del dominio en el que se desarrolla la aplicación. Pueden ser requisitos funcionales nuevos, restringir los existentes o establecer cómo se deben ejecutar cálculos particulares. Los requisitos de dominio son importantes porque es imposible que el sistema trabaje satisfactoriamente si se incumplen. Los siguientes ejemplos aclaran el concepto.

Deberá existir una interfaz del usuario estándar para todas las bases de Datos, la cual tome como referencia el estándar Z39.50. (En un dominio de biblioteca.)

La desaceleración del tren con alarma de luz roja se calculará como: $D_{min}=D_{control} + D_{gradiente}$; donde la $D_{gradiente}$ es conocida y distinta según los tipos de trenes (Dominio de control ferroviario)

La dificultad de trabajar con estos requisitos es que se expresan en un lenguaje específico de la aplicación que puede no ser bien comprendido por el analista. Además el usuario puede dejar fuera información necesaria porque para ellos es obvia.

4.2.2.- La duración del análisis.

El análisis de requisitos, debido a sus dificultades, es la fase que con más frecuencia tiende a expandirse indefinidamente en el tiempo, por tanto una de las primeras cuestiones que cabe realizar es: ¿Cuánto esfuerzo debemos dedicar al análisis?. Obviamente la respuesta dependerá de la naturaleza, el tamaño y la complejidad de la aplicación pero como guía se puede utilizar la regla de 40-20-40. En tal regla se propone que el análisis de requisitos y el diseño ocuparían el primer 40% mientras que las pruebas del sistema ocuparían el 40% final dejando el otro 20% para la codificación.

En la distribución del primer 40%, que es el que nos ocupa, la directriz que se plantea propone dedicar del 20% al 25% al diseño y entre el 15% y el 20% al análisis de requisitos. Finalmente podríamos afinar el tiempo de análisis dividiendo el tiempo entre un 10% y un 20% del esfuerzo al análisis del sistema y otro 10% o 20% al análisis de requisitos del software. Cabe destacar en este punto que no siempre será necesario realizar el análisis de sistema por lo que es posible una asignación total del 20%, máximo para el análisis, al análisis de requisitos del software

A pesar de estas consideraciones el esfuerzo que se le dedica normalmente al análisis es mucho menor. En la mayoría de los proyectos la regla no se cumple y la mayor parte del esfuerzo se va en la codificación, precisamente por la dificultad de realizar la codificación cuando no se ha hecho un buen análisis previo.

4.2.3.- El personal implicado.

En este punto lo primero que cabe plantearse es ¿quién tiene la responsabilidad de realizar el análisis?. La respuesta es fácil: el analista. Éste debe ser una persona con buena formación técnica y con experiencia. No obstante, el analista no trabaja de forma aislada sino en estrecho contacto con el personal de dirección, técnico y administrativo tanto del cliente como del que desarrolla el software.

En la actividad de análisis de requisitos el analista tiene un papel destacado en la comunicación entre Usuario/cliente y desarrolladores de software ya que, por un lado, los técnicos no conocen los detalles del trabajo de la empresa para la cual se va a desarrollar y, por otro, los usuarios no saben qué información es necesaria para el desarrollo de una aplicación.

El analista es el principal puente que se establece entre el cliente y los desarrolladores. No sólo desarrollará su trabajo en la fase de análisis de requisitos sino que lo normal es que esté presente en todas las revisiones que se hagan a lo largo del desarrollo del proyecto. En su trabajo, el analista debe entrevistarse con múltiples personas, que tendrán una visión distinta del problema y de su solución incluso tendrán intereses contrapuestos.

Un buen analista, por tanto, debe tener facilidad de comunicación, debe ser capaz de extraer información relevante a partir de fuentes confusas o contradictorias, reorganizar esta información para sintetizar soluciones y debe servir de mediador entre las partes. Además de todo esto, debe tener los conocimientos técnicos y la experiencia necesarios para poder aplicar la informática a los problemas del cliente, debe conocer o ser capaz de asimilar conocimientos sobre el campo de actividad del cliente y debe tener claros los principios básicos de la ingeniería del software para poder incorporar estos principios a los requisitos del sistema de forma que se desarrolle software de calidad.

Pero la obtención y análisis de requisitos incluyen también a diferentes tipos de personas de la organización. El término **stakeholder** se utiliza para referirse a cualquier persona que tiene influencia directa o indirecta sobre los requisitos del sistema. Entre los stakeholders se encuentran los usuarios finales que interactúan con el sistema y todos aquellos en la organización que se verán afectados por dicho sistema. Los stakeholders también son los ingenieros que desarrollan o dan mantenimiento a otros sistemas relacionados, los administradores del negocio, los expertos en el dominio del sistema, los representantes de los trabajadores, etcétera.

4.2.4.- Dificultades del análisis de requisitos.

¿Por qué es una tarea tan difícil? Básicamente porque consiste en la traducción de unas ideas vagas de necesidades de software en un conjunto concreto de funciones y restricciones. Además el analista debe extraer información dialogando con muchas personas y cada una de ellas se expresará de una forma distinta, tendrá conocimientos

informáticos y técnicos distintos, y tendrá unas necesidades y una idea del proyecto muy particulares. Somersville lo concreta en las siguientes razones:

1. Los stakeholders a menudo no conocen realmente lo que desean obtener del sistema de cómputo excepto en términos muy generales; podrían encontrar difícil articular lo que quieren del sistema; podrían hacer demandas irreales debido a que no conocen el costo de sus peticiones.

2.-Los stakeholders de un sistema expresan los requisitos con sus propios términos de forma natural y con un conocimiento implícito de su propio trabajo. Los ingenieros de requisitos, sin experiencia en el dominio del cliente deben comprender estos requisitos.

3. Diferentes stakeholders tienen requisitos distintos y podrían expresarlos de varias formas. Los ingenieros de requisitos tienen que descubrir todas las fuentes potenciales de requisitos así como las partes comunes y en conflicto.

4. Los factores políticos influyen en los requisitos del sistema. Éstos provienen de los administradores quienes solicitan requisitos específicos para el sistema debido que esto les permitirá incrementar su influencia en la organización.

5. El entorno económico y de negocios en el que se lleva a cabo el análisis es dinámico. De forma inevitable cambia durante el proceso de análisis. Por lo que la importancia de ciertos requisitos puede cambiar. Emergen nuevos requisitos de nuevos stakeholders quienes no habían sido consultados previamente.

4.3.-Recogida inicial de requisitos

Según Raghavan¹ el proceso de análisis de requisitos se debería realizar siguiendo los siguientes 5 pasos.

- **Identificar las fuentes de información** (usuarios) relevantes para el proyecto.
- **Realizar las preguntas apropiadas** para comprender sus necesidades.
- **Analizar la información** recogida para detectar los aspectos que quedan poco claros.
- **Confirmar con los usuarios** lo que parece haberse comprendido de los requisitos.
- **Sintetizar los requisitos** en un documento de especificación apropiado.

Uno de los mayores problemas que surgen es cómo poner en contacto a usuarios y técnicos para establecer unos requisitos entendibles y aceptados por todos. Con este fin se plantean distintas técnicas de comunicación y recogida de información.

Las técnicas principales utilizadas para esta actividad son las siguientes [FLAATEN et al., 1989]:

¹ Piattini Pag 155-166

- **Entrevistas.** Es quizás la técnica más empleada y la que requiere una mayor preparación (y experiencia) por parte del analista. Es similar a una entrevista periodística en la que el desarrollador entrevista uno a uno a los futuros usuarios del software.
- **Desarrollo conjunto de aplicaciones (JAD).** Se crean equipos de usuarios y analistas que se reúnen para trabajar conjuntamente en la determinación de las características que debe tener el software para satisfacer las necesidades de los usuarios. Tiene una mayor probabilidad de éxito, ya que involucra al usuario en el proyecto, que lo aprecia como algo propio. A continuación desarrollaremos en detalle la Técnica para facilitar las especificaciones de una aplicación (TFEA)
- **Prototipado.** Consiste en la construcción de un modelo o «maqueta» del sistema que permite a los usuarios evaluar mejor sus necesidades, analizando si el prototipo que ven tiene las características de la aplicación que necesitan.
- **Observación.** Consiste en analizar in situ cómo funciona la unidad o el departamento que se quiere informatizar. Aporta grandes ventajas sobre las otras técnicas, ya que se pueden analizar mejor todos los detalles del proceso y se llega a captar el funcionamiento real de la empresa (que, a veces, no coincide con las normas oficiales), así como el ambiente en el que se va a desarrollar el proyecto y se va a instalar el futuro sistema.
- **Estudio de documentación.** En casi todas las organizaciones existen documentos que describen el funcionamiento del negocio, desde planes estratégicos hasta manuales de operación. El analista debe estudiar esta documentación para hacerse una idea de la normativa que rige la empresa. También es conveniente que recopile muestras de los impresos (que deben estar necesariamente llenados para constatar que se usan realmente y para apreciar cómo se emplean) que se utilizan, ya que nos permiten conocer los datos que se manejan.
- **Cuestionarios.** Resultan útiles para recoger información de un gran número de personas en poco tiempo, especialmente en situaciones en las que se da una gran dispersión geográfica.
- **Tormenta de ideas (Brainstorming).** Algunos autores [RAGHAVAN et al., 1994] proponen la utilización de este tipo de reunión como medio para identificar un primer conjunto de requisitos en aquellos casos en los que no están muy claras todas las necesidades que hay que cubrir. Consiste en reuniones de cuatro a diez personas (usuarios) en las cuales, y en una primera fase, se sugieren toda clase de ideas sin juzgarse su validez, por muy disparatadas que parezcan. En una segunda fase, se realiza una análisis detallado de cada propuesta.
- **ETHICS8.** Constituye un método bastante evolucionado para fomentar la participación de los usuarios en los proyectos. Creado por E. Mumford en 1979 coordina la perspectiva social de los sistemas con su implementación técnica. Un sistema no tiene éxito si no se ajusta a los factores sociales y organizativos que rigen la empresa. Se busca la satisfacción de los empleados en el trabajo a través de estudios integrales (similares a los realizados por los especialistas en RR.HH. basados en factores como el conocimiento, la psicología, la eficiencia, la motivación, etc.). Los requisitos técnicos del sistema serán los necesarios para mejorar la situación de los empleados (y, por lo tanto, su productividad) en función de dichos análisis.

En la práctica es habitual utilizar combinaciones de diversas técnicas para recoger información de los usuarios. Los cuestionarios, la observación y el estudio de documentación son técnicas que, por sí solas, no suelen bastar para obtener la información necesaria para el análisis. Por eso, se suelen emplear en coordinación con las entrevistas y el TFEA, que analizaremos en los próximos apartados. El prototipado constituye una técnica bastante especial, ya que implica una forma distinta de concebir el desarrollo de software. No obstante, para crear un primer prototipo debe recurrir a las otras técnicas para tener un mínimo conocimiento de las necesidades del usuario.

4.3.1.- Método de trabajo TFEA

Mientras que los métodos clásicos se basan en entrevistas bilaterales (el analista y cada una de las partes) con lo que dejan para el analista toda la labor de organización y obtención de un consenso, últimamente se tiende a prácticas más relacionadas con el *brainstorming* en el que cada parte expone sus ideas y propuestas y se produce un debate de forma que las posiciones vayan acercándose sucesivamente hasta que se llegue a un consenso. Son las denominadas Técnicas para facilitar las especificaciones de una aplicación (TFEA)².

El método de trabajo sería el siguiente:

Inicialmente se llevan a cabo unas reuniones preliminares con el cliente (el que ha encargado el proyecto) con vistas a aclarar el ámbito del problema y los objetivos generales de una solución del mismo. Como resultado de estas reuniones, el analista y el cliente redactan una descripción breve del problema y los objetivos del proyecto y el esquema general de la solución.

A continuación se convoca una reunión en la que deben estar representados el analista, el cliente y el equipo de desarrollo. Los invitados a la reunión deben conocer previamente la descripción del problema y su solución redactada anteriormente y se les pide que elaboren una relación preliminar de los objetos o entidades que pueden identificar en el sistema o su entorno, las operaciones que se realizan o sirven para interrelacionar estos objetos, las restricciones que debe cumplir el sistema y los rendimientos que se esperan del mismo. Para conducir la reunión se nombra a un moderador neutral.

La reunión comienza discutiendo la necesidad y justificación del proyecto (es decir, debatiendo el documento preliminar). Una vez que todo el mundo esté de acuerdo en la necesidad de desarrollar el producto, se puede pasar a presentar cada una de las listas de objetos, operaciones, restricciones y rendimientos realizadas individualmente. A partir de estas listas individuales se crean listas combinadas, en las que se agrupa lo redundante pero no se elimina nada. En este punto está estrictamente prohibido el debate o las críticas.

Puede entonces comenzar la discusión sobre la lista combinada, en la que se pueden añadir nuevos elementos, eliminar otros o reescribirla de forma que refleje

² Pressman5, pag. 184

correctamente el sistema a desarrollar. Hay que realizar una *miniespecificación* de cada uno de los elementos de las listas, donde se defina brevemente cada uno de dichos elementos. El desarrollo de estas miniespecificaciones puede descubrir nuevos elementos o demostrar que alguno de ellos está incluido en otro. Se pueden dejar en el aire ciertos aspectos siempre y cuando se registre que deben ser tratados posteriormente.

Una vez concluida la reunión el analista debe elaborar un borrador de especificación del proyecto, combinando las miniespecificaciones de cada elemento, que servirá de base para todo su trabajo posterior.

Método de trabajo TFEA.

- *Redacción de un documento inicial:*
 - Descripción del problema.*
 - Objetivos.*
- *Propuesta de solución (si es posible).*
- *Creación de listas individuales de:*
 - Objetos: del sistema y del entorno*
 - Operaciones: relación entre los objetos*
 - Restricciones*
 - Rendimiento*
- *Creación de una lista conjunta*
- *Discusión de la lista conjunta*
- *Redacción de miniespecificaciones*
- *Redacción de un borrador de la especificación*

Este enfoque en equipo de la recogida inicial de requisitos proporciona numerosas ventajas, entre las que se pueden citar la comunicación multilateral de todos los involucrados en el proyecto, el refinamiento instantáneo y en equipo de los requisitos a partir de las ideas previas individuales y la obtención de un documento que sirva de base para el proceso de análisis.

Caso Práctico: Hogar Seguro

Descripción: Nuestras investigaciones indican que el mercado de sistemas de seguridad para el hogar está creciendo a un ritmo del 40% anual. Nos gustaría entrar en este mercado construyendo un sistema de seguridad para el hogar basado en un microprocesador que proteja y/o reconozca varias situaciones indeseables tales como irrusiones ilegales, fuego, inundaciones y otras. El producto, provisionalmente llamado *HogarSeguro*, utilizará los sensores adecuados para detectar cada situación, ha de ser programable por el usuario y llamará automáticamente a una agencia de vigilancia cuando se detecte alguna de estas situaciones.

EQUIPO TFEA:

Representantes de Marketing
Ingenieros de Software

Ingenieros de Hardware
Ingenieros Industriales (para la fabricación)
Coordinador

Listas de objetos:

Detectores de humos
Sensores en puertas y ventanas
Sensores de humo
Sensores de movimiento
Alarma acústica y visual
Panel de control
Pantalla
Teclado ...

Listas de acontecimientos (servicios):

Instalación de alarma
Vigilancia de sensores
Llamada de teléfono al centro de vigilancia
Programación del panel
Interacción con la pantalla de visualización
...

Restricciones:

Coste de fabricación < 200 €
Interfaz usable
Conexión a línea telefónica habitual
Conexión a línea telefónica inalámbrica
Funcionamiento mínimo independiente de electricidad
...

Rendimiento:

Respuesta a un acontecimiento < 1 seg.
Establecer listas de prioridades de actuación ante varios acontecimientos
...

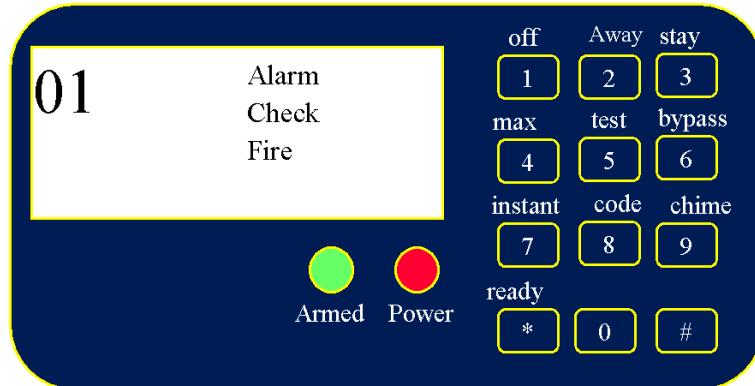
Pasos a seguir en la reunión reunión:

- 1.- Necesidad/Justificación
- 2.- Realización de listas conjuntas
- 3.- Miniespecificaciones para cada elemento de la lista
- 4.- Presentación de miniespecificaciones
- 5.- Descubrimiento de nuevos elementos
- 6.- Aspectos no abordables
- 7.- Primer borrador de especificaciones

Miniespecificación para el panel de control:

Montado en la pared

Tamaño aprox. 23x13 cm.
 Teclado estándar de 12 teclas+teclas especiales
 Pantalla LCD según gráfico adjunto
 Interacción con usuario única por teclado
 Software de ayuda para manejo
 Conexión con todos los sensores



4.4.- Análisis de requisitos del sistema

El objetivo es conseguir representar un sistema en su totalidad, incluyendo hardware, software y personas, mostrando la relación entre los diversos componentes y sin entrar en la estructura interna de los mismos. En algunos casos se nos plantearán diferentes posibilidades y tendremos que realizar un estudio de cada una de ellas.

4.4.1.- Fases del análisis de sistemas.

El análisis del sistema comprende las siguientes fases o tareas:

- *Identificación de las necesidades del cliente.*
- *Asignación de funciones a cada uno de los elementos del sistema.*
- *Evaluación de la viabilidad del sistema.*
 - *Ánálisis económico y técnico del proyecto.*
- *Obtención de una definición del sistema que sea la base del trabajo posterior.*

4.4.1.1.- FASE: Identificación de las necesidades.

Para identificar las necesidades, el analista del sistema debe reunirse con el cliente o con su representante. Juntos, proceden a definir los objetivos del sistema, la información que se va a suministrar, la información que se va a obtener, las funciones y el rendimiento requerido.

El papel del analista de sistemas es el de definir los elementos de un sistema informático dentro del contexto del sistema en que va a ser usado y debe ser capaz de distinguir entre lo que *necesita* el cliente (lo que es para él imprescindible), y lo que *quiere* el cliente (aquellos que le sería útil pero no imprescindible). El analista tiene que identificar las funciones del sistema y asignarlas a alguno de sus componentes. Para ello, parte de los objetivos y restricciones definidos por el usuario y realiza una representación de la función del sistema, de la información que maneja, de sus interfaces y del rendimiento y las restricciones del mismo.

En la mayoría de los casos, el proyecto empieza con un concepto más bien vago y ambiguo de cuál es la función deseada. Entonces el analista debe delimitar el sistema, indicando el ámbito de funcionamiento y el rendimiento deseados. Por ejemplo, en el caso de un sistema de control, no basta con decir algo así como '*el sistema debe reaccionar rápidamente en caso de que la señal de entrada supere los límites de seguridad*' sino que hay que definir cuáles son los límites de seguridad de la señal de entrada, en cuánto tiempo debe producirse la reacción y cómo ha de ser esta reacción.

Una vez que se ha logrado delimitar la función, el ámbito de información, las restricciones, el rendimiento y las interfaces del sistema, el analista debe proceder a la asignación de funciones. Las funciones del sistema deben de ser asignadas a alguno de sus componentes (ya sean éstos software, hardware o personas). El analista debe estudiar varias opciones de asignación (considerando, por ejemplo, la posibilidad de automatizar o no alguna de estas funciones), teniendo en cuenta las ventajas e inconvenientes de cada una de ellas (en cuanto a viabilidad, costes de desarrollo y funcionamiento y fiabilidad) y decidirse por una de ellas, o bien presentar un estudio razonado de las opciones a quienes tengan que tomar la decisión. Para explicar todo lo anterior podemos poner el siguiente ejemplo³:

Especificación informal del sistema de clasificación de paquetes.

El sistema de clasificación de paquetes debe realizarse de forma que los paquetes que se mueven a lo largo de una cinta transportadora sean identificados (para lo cual van provistos de un código numérico) y clasificados en alguno de los seis contenedores situados al final de la cinta. Los paquetes de cada tipo aparecen en la cinta siguiendo una distribución aleatoria y están espaciados de manera uniforme. La velocidad de la cinta debe ser tan alta como sea posible; como mínimo el sistema debe ser capaz de clasificar 10 paquetes por minuto. La carga de paquetes al principio de la cinta puede realizarse a una velocidad máxima de 20 paquetes por minuto. El sistema debe funcionar las 24 horas del día, siete días a la semana.

Función del sistema.

³ (Pressman5, pags 81 y 176)

Realizar la clasificación de paquetes que llegan por una cinta transportadora en seis compartimentos distintos, dependiendo del tipo de cada paquete.

Información que se maneja.

Los paquetes disponen de un código numérico de identificación.

Debe existir una tabla o algoritmo que asigne a cada número de paquete el contenedor donde debe ser clasificado.

Interfaces del sistema.

El sistema de clasificación se relaciona con otros dos sistemas. Por un lado tenemos la cinta transportadora. Parece conveniente que el sistema de clasificación pueda parar el funcionamiento de la cinta o del sistema de carga de paquetes en la cinta, en caso de que no pueda realizar la clasificación (por ejemplo si se produce una avería). Por otro lado, el sistema deposita paquetes en los contenedores, pero no se establece ningún mecanismo de vaciado o sustitución de los contenedores si se llenan. El sistema debería ordenar la sustitución o vaciado del contenedor o esperar mientras un contenedor esté lleno.

Como la descripción realizada por el cliente no establece los mecanismos para solventar estas dos situaciones estos detalles deben ser discutidos con el cliente.

Rendimiento.

El sistema debe ser capaz de clasificar al menos 10 paquetes por minuto. No es necesario que el sistema sea capaz de clasificar más de 20 paquetes por minuto.

Restricciones.

El sistema debe tener un funcionamiento continuo. Por tanto, debemos evitar la parada del sistema incluso en el caso de que para alguno de los componentes del mismo se averíen.

El documento no indica restricciones sobre la eficacia del sistema, es decir, sobre cuál es el porcentaje máximo que se puede tolerar de paquetes que pueden ser clasificados de forma errónea. Estos detalles también deben ser aclarados con el cliente.

Asignación de funciones.

Podemos considerar tres asignaciones posibles:

Asignación 1.

*Esta asignación propone una **solución manual** para implementar el sistema. Los recursos que utiliza son básicamente personas, y se requiere además algo de documentación, definiendo las características del puesto de trabajo y del sistema de turnos y una tabla que sirva al operador para relacionar los códigos de identificación de los paquetes con el contenedor donde deben ser depositados.*

La inversión necesaria para poner en marcha este sistema es mínima, pero requiere una gran cantidad de mano de obra (varios turnos de trabajo y operadores de guardia) con lo que los costes de funcionamiento serán elevados. Además hay que tener en cuenta que lo rutinario del trabajo provocará una falta de motivación en los operarios, lo que a la larga se acabará traduciendo en un mayor absentismo laboral. Todos estos factores deben de tenerse en cuenta a la hora de elegir esta u otra opción.

Asignación 2.

*En este caso, la solución es automatizada. Los recursos que se utilizan son: hardware (el lector de códigos de barras, el controlador, el **mecanismo de distribución**), software (para el lector de códigos de barras y el controlador, y una base de datos que permita asignar a cada código su contenedor) y personas (si en caso de avería la distribución se va a hacer manualmente). Cualquiera de los elementos hardware y software tendrán la correspondiente documentación sobre cómo han sido construidos y un manual de usuario.*

Aquí si hay que realizar una cierta inversión, para comprar o desarrollar los componentes del sistema, pero los costes de funcionamiento serán sin duda menores (sólo el consumo de energía eléctrica). Hay que tener en cuenta que el uso de dispositivos mecánicos (el mecanismo de distribución) va a introducir unos costes de mantenimiento y paradas por avería o mantenimiento con una cierta frecuencia.

Asignación 3.

*Los recursos que utilizamos aquí son: hardware (el lector, el **brazo robot**), software (el del lector, el del robot, incluyendo la tabla o algoritmo de clasificación) y la documentación y manuales correspondientes a estos elementos.*

En este caso la inversión inicial es, sin duda, la más elevada. Los costes de funcionamiento son bajos pero hay que considerar también el coste de mantenimiento del robot, que posiblemente tenga que ser realizado por personal especializado. Los

únicos motivos que nos harían decidir por esta opción en vez de la anterior vendrían dados por una mayor velocidad, un menor número de errores o unas menores necesidades de mantenimiento o frecuencia de averías.

Por otra parte esta solución puede tener problemas de viabilidad (si no encontramos un brazo robot que sea capaz de atrapar los paquetes según pasan por la cinta).

Además de las tres opciones propuestas, el ingeniero de sistemas debe considerar también la adopción de *soluciones estándar* al problema. Hay que estudiar si existe ya un producto comercial que realice la función requerida para el sistema o si alguna de las partes del mismo pueden ser adquiridas a un tercero. Aparte de considerar el precio de estos productos habrá que tener también en cuenta los costes del mantenimiento y el riesgo que se asocia al depender de una tecnología que no es propia (¿es la empresa proveedora estable?, ¿cuál es la calidad de sus productos?) valorando todo esto frente a los riesgos asociados a realizar el desarrollo nosotros mismos.

La labor del ingeniero o analista de sistemas consiste, en definitiva, en asignar a cada elemento del sistema un ámbito de funcionamiento y de rendimiento. Después, el ingeniero del software se encargará de refinar este ámbito para el componente software del sistema y de producir un elemento funcional, que sea capaz de ser integrado con el resto de los elementos del sistema.

4.4.1.2.- FASE: Asignación de funciones.

Como ya habíamos visto, para cada elemento del sistema puede haber una o varias asignaciones posibles. Es necesario estudiar cada una de las alternativas desde el punto de vista económico y técnico y elegir cuál es la más adecuada.

Para elegir entre las distintas alternativas debemos fijar una serie de parámetros de evaluación. Normalmente los parámetros de evaluación serán de orden económico (coste de las inversiones, ahorro que se producirá con el sistema nuevo), pero también pueden tenerse en cuenta parámetros relacionados con la fiabilidad del sistema, su capacidad, su rendimiento o la mejora de calidad de los productos, si se trata de un sistema de producción. Hay que ordenar cada uno de estos parámetros de acuerdo a su importancia, lo que dependerá de los objetivos generales de cada proyecto, y ver cuál de las distintas alternativas obtiene una mejor evaluación de acuerdo con los parámetros establecidos. Cuando dos o más alternativas satisfagan los parámetros de evaluación principales, optaremos entre ellas observando los parámetros de segundo orden y así sucesivamente.

Por ejemplo, en el caso del sistema de clasificación de paquetes, la alternativa elegida dependerá de qué parámetros consideremos más importantes. Si queremos que la inversión inicial sea pequeña la mejor alternativa será la primera. Si pretendemos en cambio unos costes de funcionamiento moderados, valorando que el plazo de amortización sea corto, la mejor opción será la segunda. Si damos más importancia a la fiabilidad y al bajo mantenimiento del sistema, la mejor opción puede ser la tercera.

Sistema de clasificación de paquetes. Criterios de evaluación.

	Alternativa 1	Alternativa 2	Alternativa 3
Inversión inicial	Nula	Moderada	Grande
Coste funcionamiento	Grande	Moderado	Moderado
Tiempo amortización	Nulo	Moderado	Grande
Fiabilidad	Moderada	Pequeña	Grande
Mantenimiento	Nulo	Grande	Moderado

4.4.2.- Estudio de viabilidad.

Cualquier proyecto sería viable si dispusiésemos de recursos humanos, temporales y económicos ilimitados. Pero los recursos son siempre limitados: existen restricciones sobre el número de personas que se pueden dedicar (especialmente si se trata de personal del cliente), sobre cuánto dinero nos podemos gastar en el proyecto (si la inversión necesaria para el desarrollo es demasiado alta el sistema no compensará los ahorros que se produzcan con su uso) y sobre los tiempos de entrega (nadie compra software a cinco años vista, además, según pasa el tiempo aumentan las posibilidades de que cambien los requisitos).

Por esto, es conveniente estudiar la viabilidad del proyecto lo antes posible, puesto que así se pueden ahorrar meses de esfuerzos y miles de euros en el desarrollo de un proyecto que al final se muestre como inviable. El estudio de viabilidad está muy relacionado con el análisis de riesgos: si el riesgo de un proyecto es grande, se reducen las posibilidades de producir software de calidad, es decir, disminuye la viabilidad.

Viabilidad económica. Consiste en comparar los beneficios futuros de la utilización del sistema con los costes de su desarrollo. La justificación económica es normalmente la principal consideración a la hora de decidir realizar o no cualquier proyecto. Por esto, aparte de decidir la viabilidad o no viabilidad se necesita también un análisis económico completo (no sólo si va a producir beneficios sino qué beneficios va a producir, a qué plazo, etc.).

Viabilidad técnica. Consiste en determinar si es posible desarrollar o no el producto, teniendo en cuenta sus restricciones, basándonos en los recursos humanos y técnicos a nuestro alcance. Como los objetivos, funciones y rendimiento son confusos al inicio del proyecto, cualquier cosa puede parecer inicialmente viable. Debido a esto, es conveniente revisar el estudio de viabilidad técnica una vez que las especificaciones estén más claras. Además, el análisis de viabilidad técnica debe ser completado con un estudio técnico del sistema en proyecto, que permita determinar las características técnicas del nuevo sistema y qué mejoras introduce sobre el proceso actual.

Viabilidad legal. Consiste en determinar si el proyecto infringe alguna disposición legal sobre el derecho a la intimidad de las personas, las normas de seguridad en el trabajo, de calidad de los productos, las leyes de Copyright si se van a utilizar componentes comprados a terceros para integrarlos en el sistema o basarnos en ellos para el desarrollo del producto software, y otras.

Viabilidad de Plazos. Estudiada en el tema anterior se trata de calcular los plazos para la realización viable del proyecto o comprobar si los plazos a los que se ha comprometido quien firma el contrato permiten realizar una calendarización realista.

4.4.2.1.- Análisis económico.

El análisis económico del proyecto consistirá en señalar los costes de desarrollo del proyecto y compararlos con los beneficios que producirá una vez desarrollado.

Los costes de desarrollo pueden ser cuantificados fácilmente (inversiones en equipos, horas-hombre necesarias en las fases de análisis, diseño y codificación). Sin embargo, suele ser más difícil determinar los beneficios futuros que producirá el proyecto una vez listo para ser usado. Algunos de estos beneficios pueden ser tangibles (disminución del tiempo necesario para realizar determinadas tareas) pero otros muchos son intangibles (mayor satisfacción o cualificación profesional de los usuarios del sistema, incremento de la capacidad de gestión, etc.), por lo que puede ser difícil realizar comparaciones coste-beneficio directas.

Hay que tener en cuenta que la mayoría de los sistemas de procesamiento de la información se realizan teniendo como principal objetivo una mayor cantidad, calidad y rapidez de acceso a la información, de forma que se pueda realizar una mejor gestión de la empresa. Todos estos beneficios citados son intangibles y, aunque se traducirán sin lugar a dudas en beneficios tangibles (ahorro en los costes de producción o en las tareas administrativas, incremento de ventas como resultado de una mayor rapidez de respuesta a las necesidades del mercado, etc.) es muy difícil formular numéricamente esta relación. Por este motivo, los beneficios intangibles se incluirán en el análisis económico para reforzar la justificación del proyecto, pero esta justificación debe basarse en la medida de lo posible en los beneficios, medibles y demostrables que producirá el nuevo sistema.

Los costes y beneficios pueden ser directos o indirectos dependiendo de si es posible establecer una relación de los mismos con la implantación del software. Por ejemplo un aumento de la capacidad de gestión de un almacén es un beneficio directo de la implantación de un sistema informático para el control del mismo. El acondicionamiento del almacén para adaptarlo al nuevo sistema (redistribución de espacios, etc.) puede considerarse un coste indirecto.

La única forma de demostrar estos beneficios será comparando los modos de trabajo con el nuevo sistema con los modos de trabajo actuales. El nuevo sistema, cambiará los modos de trabajo de la empresa u organización en la que se instale de forma que se producirá:

- una disminución del tiempo necesario para realizar determinadas tareas.
- una menor necesidad de mano de obra para realizar el trabajo actual.
- un aumento de productividad, que permitirá aumentar la producción con la mano de obra actual.

Cualquiera de los tres puntos anteriores puede ser evaluado cuantitativamente, de forma que se puede determinar el ahorro que se producirá con la puesta en marcha del nuevo sistema.

Pongamos como ejemplo el sistema de clasificación de paquetes visto antes. Supongamos que el sistema actual es el que corresponde a la *Asignación 1* (es decir, es un sistema manual) y que el nuevo sistema es el indicado en la *Asignación 3* (utilización de un lápiz óptico y de un brazo robot).

Sistema de clasificación de paquetes. Análisis coste-beneficio

Costes del sistema actual: 105.000 €/año

Costes del nuevo sistema:

Inversión inicial: 240.000 €

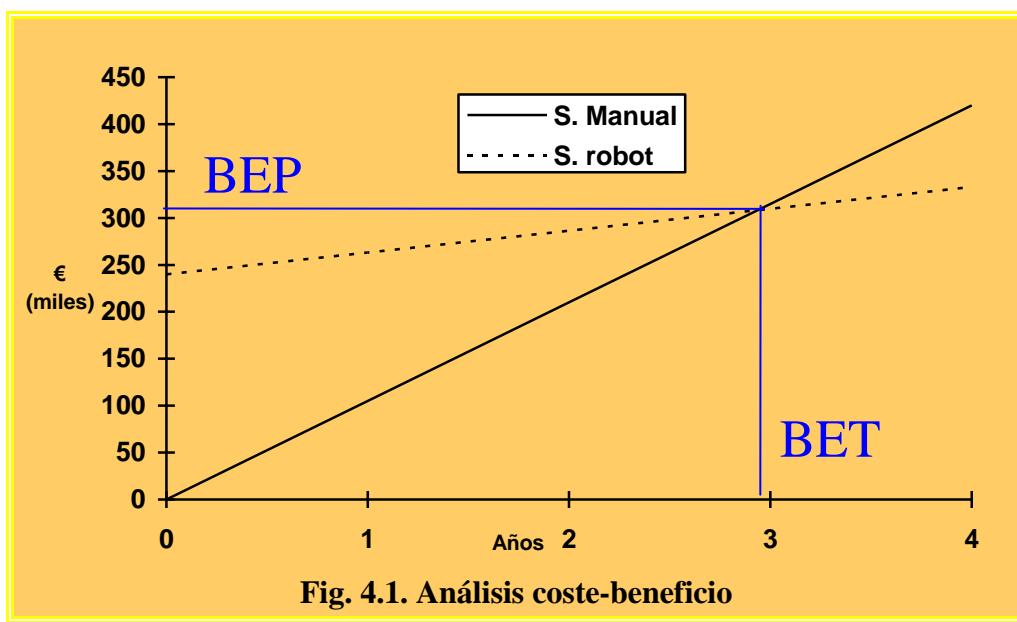
Funcionamiento: 12.000 €/año

Mantenimiento: 6.000 €/año

Operatividad: 95 %

Total coste anual: 23.256 €/año

En primer lugar, calculemos los costes del sistema actual. Supongamos que para el funcionamiento en turno ininterrumpido, incluyendo operadores de guardia, necesitamos 7 operadores y que el coste salarial (incluyendo cargas sociales) de cada operador es de 15.000 €/año. Podemos calcular el coste/hora del sistema actual, que resulta ser de 12 €/hora, aproximadamente.



Supongamos también que las inversiones necesarias para poner en marcha el sistema nuevo son de 240.000 €, que los costes de mantenimiento del brazo robot son de 6.000 €/año y los costes de funcionamiento (gastos de electricidad) son de 12.000 €/año. Además, debido a las paradas por mantenimiento y averías, el sistema nuevo solo estará útil el 95% del tiempo, debiendo efectuarse la clasificación manual el 5% del tiempo restante. Esto resulta en unos gastos de puesta en marcha de 240.000 € y unos gastos de funcionamiento de 23.256 €/año. Podemos representar los beneficios y costes acumulados del nuevo sistema en una gráfica (fig. 4.1) y calcular el tiempo de amortización (el punto donde se cortan ambas curvas: indica cuánto tiempo ha de pasar para que los beneficios del sistema nuevo compensen los costes iniciales de puesta en marcha). Este tiempo de amortización resulta ser de 2.9 años, aproximadamente.

4.4.2.2.- Análisis técnico.

Con el análisis técnico se pretende estudiar las características técnicas del nuevo sistema: capacidad, rendimiento, fiabilidad, seguridad, etc., de forma que se complemente el análisis de coste-beneficio con las mejoras técnicas que pueda proporcionar el sistema a los modos de trabajo de la empresa u organización donde se implante.

Para hacer un buen análisis técnico tendremos que modelar el sistema de alguna forma, y realizar un estudio analítico de las características del modelo propuesto o bien realizar algún tipo de simulación con el modelo.

Los resultados del análisis técnico son otro de los criterios que permitirán decidir si seguir o no con el proyecto. Si el riesgo técnico es alto, o si el modelo o las simulaciones muestran que el sistema en proyecto no va a conseguir substanciales mejoras sobre el sistema actual, podemos cancelar el proyecto.

4.4.3.- Representación de la arquitectura del sistema.

Como parte de los requisitos y diseño del sistema, éste tiene que modelarse como un conjunto de componentes y relaciones entre ellos que sirva de base para el trabajo posterior. Para ello se utiliza comúnmente una representación en diagrama de bloques que muestre los principales subsistemas y la interconexión entre ellos. Para estandarizar esta representación Pressman nos propone el uso de los diagramas de arquitectura a los que divide en cinco regiones⁴.

⁴ (Pressman5, pags 175 y 176)

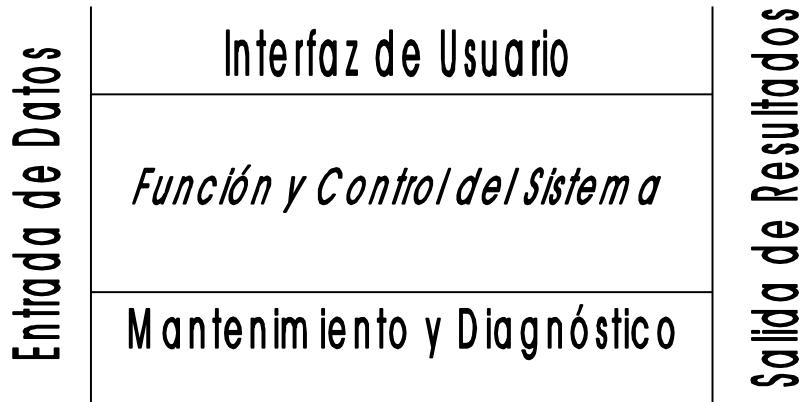


Figura 4.2.- Diagrama de arquitectura de un sistema

De esta forma los diagramas de arquitectura nos permitirán, además de representar las partes del sistema, identificar su entorno distinguiendo claramente sus interfaces externas. Si la complejidad del sistema así lo aconseja podemos utilizar estos diagramas formando una jerarquía de niveles, en el nivel superior representaremos el sistema mediante un diagrama de contexto, e iremos detallando más la arquitectura en sucesivos diagramas de flujo⁵.

Diagrama de contexto.

El diagrama de contexto representa el sistema en relación con su entorno. Sirve para definir los límites del sistema y muestra todos los productores y consumidores de información del sistema.

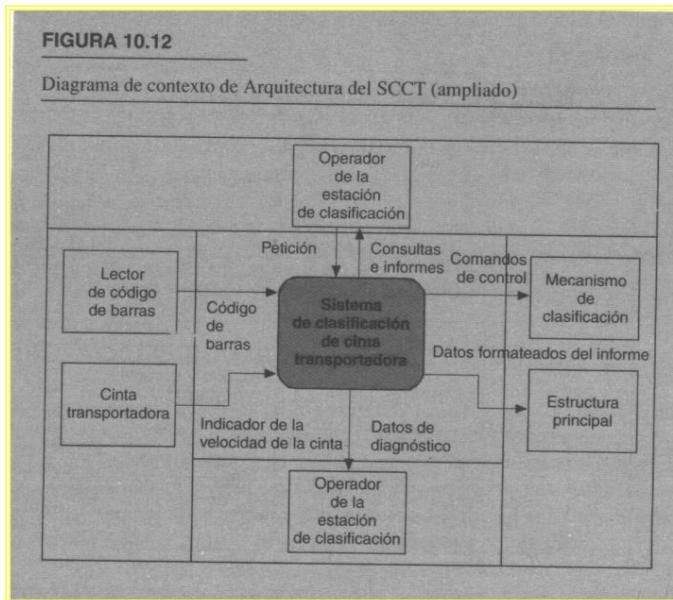


Figura 4.3.- Diagrama de contexto del sistema clasificador de paquetes

⁵ Esta metodología se desarrollará en el tema 5.

El centro del diagrama de contexto estará ocupado por el sistema, representado por una caja de esquinas redondeadas. A su alrededor se situarán una serie de entidades o agentes externos (el entorno de sistema) representados mediante cajas de esquinas cuadradas. Cada agente externo representa un productor o consumidor de información del sistema. Cada agente externo se sitúa en la región del diagrama que le corresponda, según su papel sea el de productor, consumidor, usuario o supervisor.

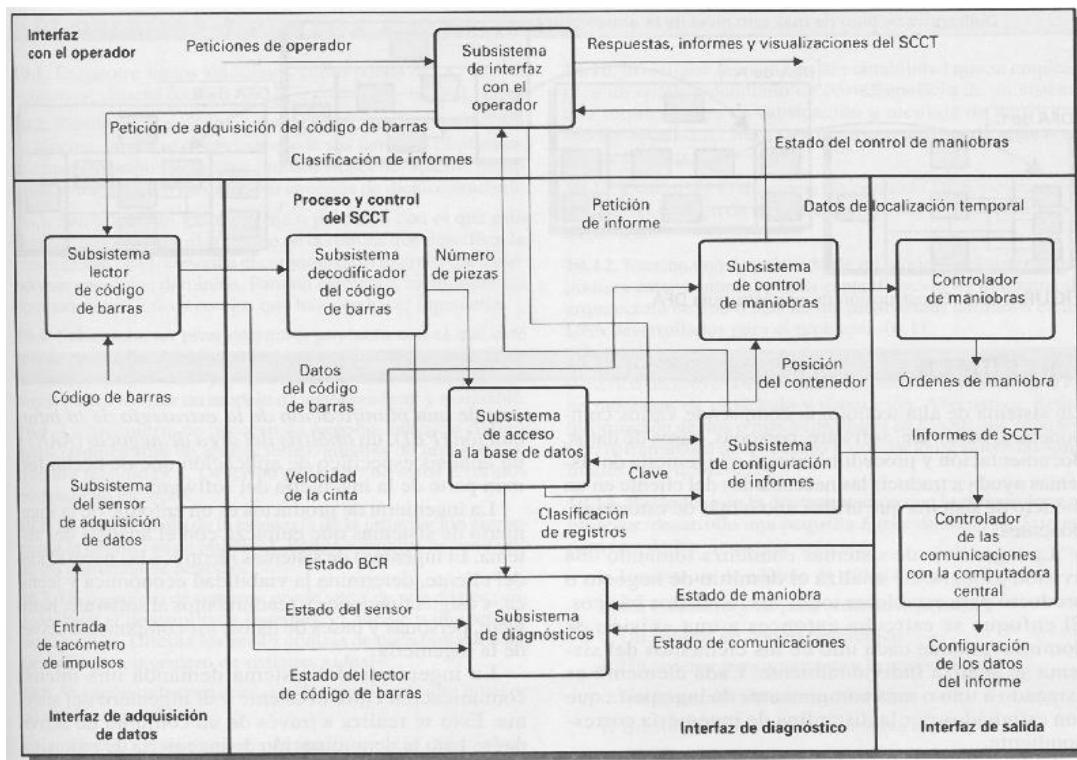
El sistema se relaciona con los agentes externos a través de flujos de información, representados mediante arcos orientados.

Diagramas de flujo.

Podemos describir la arquitectura del sistema en mayor grado de detalle a base de expandir el diagrama de contexto en una jerarquía de diagramas de flujo.

Cada subsistema del diagrama de contexto puede dar lugar a un diagrama de flujo, donde se describirá el sistema en mayor detalle, descomponiéndolo en nuevos subsistemas relacionados mediante flujos de información. Cada subsistema ocupa una región del diagrama dependiendo de cuál sea su función (adquisición de datos, salida de datos, interfaz, etc.).

Un detalle importante es que debe mantenerse la consistencia entre los diagramas de distinto nivel. Al expandir un determinado subsistema en un diagrama de flujo, los flujos de información que conectan dicho subsistema con otros o con agentes externos, deben figurar también (coincidiendo en número, sentido y nombre) en el diagrama de flujo resultado de dicha expansión. Estos flujos de información, tendrán un extremo libre, el que los conectaba con subsistemas que quedan fuera del ámbito del nuevo diagrama de flujo.



4.4.4.- Especificación del sistema.

La especificación del sistema es un documento que sirve de base para el análisis de cada uno de los componentes que intervienen en él, sean hardware, software o personas. Describe la función, rendimiento y restricciones que debe cumplir el sistema y limita cada uno de sus componentes, indicando el papel de cada uno de ellos y su interfaz.

La especificación del sistema es el documento que permite comprobar si el análisis realizado por el ingeniero de sistemas satisface las necesidades del cliente. Por este motivo, el cliente y el analista de sistemas deben revisar conjuntamente esta especificación para determinar si:

Evaluación inicial de la especificación

- *se ha delimitado correctamente el ámbito del proyecto.*
- *se ha definido correctamente la funcionalidad, las interfaces y el rendimiento.*
- *las necesidades del usuario y el análisis de riesgos justifican el desarrollo del proyecto.*
- *cliente y analista tienen la misma percepción de los objetivos del proyecto.*

Después de esta revisión con el usuario, es necesario realizar una evaluación técnica, para determinar si:

Evaluación técnica de la especificación

- *Las estimaciones de riesgos, coste y agenda se corresponden con la complejidad del proyecto.*
- *Todos los detalles técnicos (asignación de funciones, interfaces, rendimientos) están bien definidos.*
- *La especificación del sistema sirve de base para las fases siguientes (en concreto para la ingeniería de requisitos del software).*

Un posible formato de especificación del sistema podría ser el siguiente:

Especificación de requisitos del sistema.

I. Introducción.

- A. Ámbito y propósito del documento.
- B. Descripción general.
 1. Objetivos.
 2. Restricciones.

II. Descripción funcional y de datos.

- A. Diagrama de contexto de arquitectura.
- B. Descripción del DCA.

III. Descripción de los subsistemas.

- A. Especificación del diagrama de arquitectura para el subsistema *i*.
 1. Diagrama de flujo de arquitectura.
 2. Narrativa del módulo del sistema.
 3. Rendimiento.
 4. Restricciones de diseño.
 5. Asignación de componentes.
- B. Diccionario de la arquitectura.

IV. Resultados de la simulación del sistema.

- A. Modelo usado para la simulación.
- B. Resultados de la simulación.
- C. Aspectos especiales de rendimiento.

V. Aspectos del proyecto.

- A. Costes del proyecto.
- B. Agenda.
- C. Análisis de Viabilidad

VI. Apéndices.

4.5.- Análisis de requisitos del software

Como resultado de la fase de análisis de requisitos del sistema, se ha asignado una función y un rendimiento al componente software del mismo. Para conseguir esta función y rendimiento, el ingeniero del software debe construir - o adquirir - una serie de componentes software. Desgraciadamente, los componentes software están muy poco estandarizados, por lo que las dos únicas opciones serán el adquirir un sistema software comercial que cumpla con los requisitos - si este sistema existe y logramos encontrarlo - o desarrollar (o encargar el desarrollo) un sistema software a medida.

4.5.1.- Objetivos y actividades del análisis de requisitos del software.

El análisis de requisitos del software tiene como objeto desarrollar una representación del software que pueda ser revisada y aprobada por el cliente y es una tarea que sirve de enlace entre la asignación de funciones al software que se ha hecho en el análisis del sistema y el diseño del software.

Desde el punto de vista del analista de sistemas, el análisis de requisitos del software define con mayor precisión las funciones y rendimiento del software, las interfaces que ha de tener con otros componentes del sistema y las restricciones que debe cumplir.

Desde el punto de vista del diseñador, el análisis de requisitos proporciona una representación de la información, la función y el comportamiento del sistema que él se encargará de traducir en un diseño de datos y programas.

Por último, el análisis de requisitos, incluido en la especificación del proyecto, permite a todos (incluido aquí el cliente) valorar la calidad del software una vez que haya sido construido.

La labor del analista debe centrarse en el **qué**, no en el **cómo**. (¿qué datos debe manejar el sistema software?, ¿qué función debe realizar?, ¿qué interfaces debe tener?, y ¿qué restricciones tiene que cumplir?)

Pressman (Pressman 5^a 182-183) propone que el análisis de requisitos del software puede dividirse en cinco áreas de esfuerzo o actividades: (1) reconocimiento del problema, (2) evaluación y síntesis, (3) modelado, (4) especificación y (5) revisión.

Inicialmente, el, ingeniero del software o analista, estudia la especificación resultado del análisis del sistema, con el objetivo de comprender cuál es el papel del software en el contexto del sistema. El objetivo del analista es el reconocimiento de los elementos básicos del problema tal y como los percibe el cliente/usuario. Con este objetivo debe ponerse en contacto con el equipo técnico y de gestión del cliente, para poder conocer los objetivos básicos del software tal como los entiende el cliente. Con la información obtenida de la especificación del sistema y de las entrevistas con el cliente/usuario el analista debe ir definiendo y refinando los flujos y la estructura de la

información, la función del programa y su papel en el contexto del sistema, las características de las interfaces y las restricciones de diseño.

Con este conocimiento que se va adquiriendo del sistema software, el analista debe sintetizar una o varias soluciones al problema (modelos del problema), comprobando que se ajustan al plan del proyecto y a las necesidades del cliente. Este proceso de análisis del problema y síntesis de soluciones (modelos) debe proseguir hasta que el analista y el cliente acuerden una solución y se pueda especificar el software de forma adecuada para que se puedan efectuar las siguientes fases de desarrollo.

La síntesis de modelos permite entender mejor los flujos de información y la función de cada elemento del sistema software además estos modelos servirán de base para el diseño de software. A partir de estos modelos pueden desarrollarse prototipos del sistema software. Esto estará especialmente indicado cuando el cliente no esté seguro de lo que quiere realmente o cuando el analista necesite comprobar si una solución determinada resuelve efectivamente el problema. Los prototipos pueden ser utilizados por el cliente para refinar los requisitos o comprobar la validez de la solución propuesta.

La especificación debe indicar qué es lo que hay que hacer, pero no cómo hay que hacerlo. Debe ser una descripción de las necesidades y no una propuesta de solución. El cliente debe indicar qué características son imprescindibles y cuáles opcionales, y debe evitar describir la estructura interna del sistema, para no reducir la flexibilidad en la implementación. Características como, rendimiento, protocolos de comunicación, estándares de la IS (construcción modular, previsión de expansiones futuras, etc.), y, en algunos casos, la elección del soporte hardware y el lenguaje de implementación, pueden figurar en esta especificación. Otras decisiones de diseño, como el uso de un determinado algoritmo, no tienen nada que ver con el análisis y no son propiamente requisitos del sistema.

La especificación preliminar no es un documento inmutable. Normalmente será ambiguo, incompleto, incorrecto e inconsistente. Incluso aunque estén expresados de forma precisa, algunos de los requisitos reflejados pueden causar efectos secundarios no deseados (gran cantidad de información almacenada, tiempos de carga o de ejecución muy grandes) o llevar a costes de implementación demasiado grandes, frente a unos beneficios pequeños o innecesarios. Es necesario tomar la decisión de cumplir o no estos requisitos.

Otra manera de describir las actividades a realizar en la fase de análisis de requisitos está recogida por Piattini (pp 171) de Raghavan

1. **Extracción o determinación de requisitos.** El proceso mediante el cual los clientes o los futuros usuarios del software descubren, revelan, articulan y comprenden los requisitos que desean.
2. **Análisis de requisitos.** El proceso de razonamiento sobre los requisitos obtenidos en la etapa anterior, detectando y resolviendo posibles inconsistencias o conflictos, coordinando los requisitos relacionados entre sí, etc.

3. **Especificación de requisitos.** El proceso de redacción o registro de los requisitos. Para este proceso. puede recurrirse al lenguaje natural, lenguajes formales, modelos, gráficos, etc.
4. **Validación de los requisitos.** El proceso de confirmación, por parte de los usuarios o del cliente, de que los requisitos especificados son válidos, consistentes, completos, etc.

Igual que en la discusión anterior se precisa en este caso que estas actividades no tienen que realizarse en secuencia y que de hecho, habrá continuas iteraciones entre ellas. En cualquier caso la realización de estas actividades se apoya en distintas técnicas, algunas de ellas, como las de recogida de información ya han sido vistas pero la mayoría se discutirán en adelante.

4.5.2.- Principios del análisis de requisitos del software.

A lo largo de la historia de la Ingeniería del Software se han ido desarrollando diversos métodos de análisis del software, y diversas herramientas para facilitar el uso de estos métodos. Cada uno de los métodos tiene sus peculiaridades, utiliza una notación gráfica o textual distinta y tiene un campo de aplicación distinto. Sin embargo todos se basan en un conjunto de principios fundamentales.

Principios de análisis de requisitos del software

- *Identificar y representar el ámbito de información del problema.*
- *Modelar la información, la función y el comportamiento del sistema.*
- *Descomponer el problema de forma que se reduzca la complejidad.*
- *Avanzar desde lo más general a lo más detallado*

Los dos primeros puntos se refieren, por tanto, a conseguir representar el sistema y la información que maneja mediante un diagrama o una representación textual que permita comprender fácilmente qué información se maneja y qué funciones se realizan con esta información.

Los dos últimos se refieren a un método de trabajo *top-down*, que permita la división del problema en subproblemas, y vaya avanzando de los más general a lo más específico, de forma que la síntesis de la solución siga una estructura jerárquica.

El ámbito de información.

La tarea que realiza el software va a consistir siempre en procesar información: cualquier aplicación que consideremos responde a una estructura de entrada-procesamiento-salida, donde el software se encarga de procesar unos determinados datos de entrada para producir unos datos de salida. Estos datos pueden ser lógicos, numéricos, cadenas de caracteres, imágenes...

Pero en un sistema software, la información no está representada sólo por datos sino también por sucesos o eventos. En el sistema de clasificación de paquetes se procesa el número de identificación de cada paquete, obteniéndose el contenedor donde debe ser almacenado. Pero también se procesan sucesos: la clasificación de paquetes se

para si sucede que los contenedores se llenan; o se genera un informe de los paquetes procesados si el operador lo solicita. Los eventos son normalmente datos de tipo lógico (la activación o desactivación de la señal de un sensor, por ejemplo) y su procesamiento está ligado con el control del sistema.

En un sentido general, podemos decir que la información que maneja un sistema software se divide en datos y eventos. Los datos se refieren normalmente a la información que procesa el sistema y los eventos a cuándo debe procesarse esa información. El ámbito de información del sistema admite, por tanto, dos puntos de vista: por un lado, el flujo de datos (cómo se mueven los datos por el sistema y cómo se van transformando estos datos), y por otro, el flujo de control (cómo se controla el sistema y cuándo hay que realizar cada procesamiento).

En ambos casos se requiere el examen del dominio de la información y la creación de un modelo de datos que tiene tres aspectos: El contenido de la información, su estructura y el flujo de la información. El contenido hace referencia a los datos individuales que se manejan; la estructura, a como estos datos están organizados en entidades y como éstas están relacionadas; y el flujo de la información debe describir como cambian los datos y el control a medida que se mueven dentro del sistema lo que también incluye los flujos de control.

Modelado del sistema software.

Los modelos se utilizan en el campo de la ingeniería para entender mejor lo que se quiere construir. En la Ingeniería del Software se utilizan modelos para la información que transforma el software (modelos de datos), para las funciones que transforman esa información (modelos de procesos) y también para definir el comportamiento del sistema (modelos de control). Todos los métodos de análisis que veremos posteriormente son en realidad métodos de modelado de sistemas.

Los modelos que se realizan en la fase de análisis sirven para tres cosas:

Utilidad de los modelos

- *Ayudan al analista a entender la información, la función y el comportamiento del sistema, con lo que el análisis puede hacerse de forma más fácil y sistemática.*
- *Sirven de base para el trabajo del diseñador. La arquitectura de las aplicaciones y los datos debe corresponderse con los modelos del análisis.*
- *Sirven también para realizar la validación del producto una vez desarrollado. Por una parte, el sistema final debe comportarse como indica el modelo. Por otra, el sistema final permite comprobar la consistencia y la eficacia de la especificación.*

Para realizar un modelo podemos utilizar una notación gráfica (y el modelo estará representado mediante una serie de diagramas) o bien una notación textual (y tendremos modelos en lenguaje natural o en un lenguaje de especificación formal).

Descomposición del sistema.

La descomposición se utiliza para abordar problemas que son demasiado grandes o demasiado complejos para resolverlos directamente. Mediante

descomposición, el problema se divide en subproblemas más sencillos, que realizan una función más clara y que pueden entenderse más fácilmente. La descomposición se puede aplicar a los ámbitos de la información, de la función o del comportamiento.

Análisis de lo general a lo específico.

Descomponiendo un problema en subproblemas, y definiendo modelos de cada uno de estos problemas y subproblemas, llegaremos a establecer una jerarquía de modelos, que irán desde el más general (el de arriba del todo) a los más específicos (en los niveles bajos de la jerarquía). Cada modelo de la jerarquía produce, por descomposición de las funciones que realiza, una serie de modelos, cada uno de los cuales realiza una de estas funciones y que tendrán un mayor nivel de detalle.

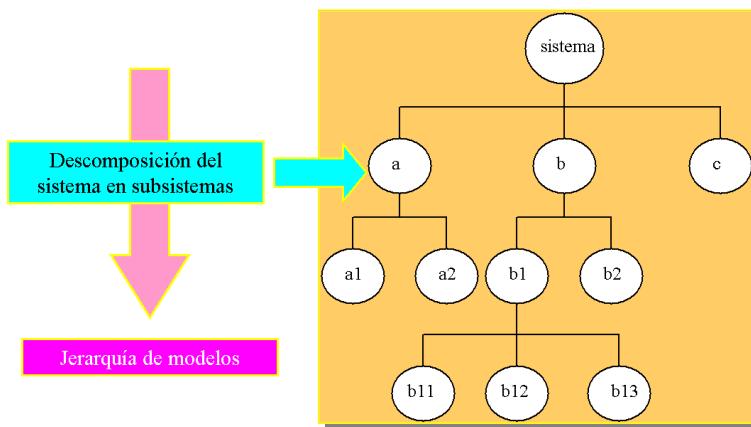


Figura 4.4.- Jerarquía de modelos

4.6.- Especificación de requisitos.

Para comprender qué es una Especificación, vamos a definir primero los siguientes términos [IEEE, 1990]:

- *Especificación es un documento que define, de forma completa, precisa y verificable, los requisitos, el diseño, el comportamiento u otras características de un sistema o componente de un sistema.*

La especificación del software es el documento que culmina las labores del análisis de requisitos. Debe contener una descripción detallada del ámbito de información, las funciones y el comportamiento asignados al software durante el análisis del sistema. Además debe contener información sobre los requisitos de rendimiento y restricciones de diseño y sobre las pruebas que se han de realizar para probar el software una vez que haya sido construido.

Por tanto, la Especificación se puede definir como la documentación de los requisitos esenciales (funciones, rendimiento, diseño, restricciones y atributos) del software y de sus interfaces externas [IEEE, 1990]. Los requisitos se representan de forma que conduzcan a una correcta implementación del sistema.

4.6.1.- Principios de la especificación

Pressman nos propone una serie de principios a seguir para la especificación:

Principios de la especificación.

- *La especificación debe modelar el dominio del problema.*
- *Es necesario separar funcionalidad e implementación.*
- *El lenguaje de especificación debe estar orientado al proceso.*
- *La especificación debe abarcar todo el sistema del que el software es parte.*
- *La especificación debe abarcar también el entorno del sistema.*
- *La especificación debe ser operativa.*
- *La especificación debe ser ampliable y tolerante a la incompletitud.*
- *La especificación debe estar localizada y débilmente acoplada.*

- **La especificación debe modelar el dominio del problema.**

La especificación del sistema ha de ser un modelo del dominio del problema, en vez de un modelo de diseño o implementación. Debe describir el sistema tal como es percibido por los expertos en el dominio de aplicación. Los elementos del sistema deben corresponderse con objetos reales de dicho dominio, ya sean individuos, máquinas u organizaciones, y las acciones que se realizan deben corresponderse con lo que realmente ocurre en el dominio del problema. Además, deben poder describirse en la especificación las reglas o leyes que gobiernan los objetos del dominio de aplicación. Algunas de estas leyes son restricciones sobre los estados del sistema, del tipo de ‘dos objetos no pueden estar en el mismo lugar al mismo tiempo’ y otras describen cómo responden los objetos cuando se actúa sobre ellos (por ejemplo las leyes del movimiento de Newton). En este caso, son parte inherente de las especificaciones del sistema.

Para que esto sea posible, el formato de la especificación y su contenido deben ser adecuados al problema.

- **Es necesario separar funcionalidad e implementación.**

Una especificación es, por definición, una descripción de lo que se quiere realizar, no de cómo se va a realizar o implementar. Como ejemplo de esto podemos tomar una especificación formal expresada mediante algún lenguaje declarativo: dado un conjunto de valores de entrada se produce otro de salida. Estas especificaciones se centran exclusivamente en el *qué* y no en el *cómo*. Esto es debido a que el resultado es una función matemática de la entrada. En estos casos de lo que se trata es de buscar alguna o todas las soluciones (modelos), tales que se cumpla $P(\text{entrada})$, donde P es un predicado que representa al sistema.

- **El lenguaje de especificación debe estar orientado al proceso.**

El ejemplo anterior muestra cómo se modelan sistemas que no están afectados por el entorno. Sin embargo, un caso más general debe considerar un sistema que interactúe con un entorno dinámico, modificando su comportamiento según los estímulos que recibe. Este podría ser el caso de un sistema empotrado. En este caso no podemos representar el sistema mediante una función matemática que relacione la entrada y la salida, sino que debemos emplear una descripción orientada al proceso, en la que la especificación del *qué* se consigue estableciendo un modelo del comportamiento deseado en términos de respuestas funcionales a distintos estímulos del entorno.

- **La especificación debe abarcar todo el sistema del que el software es parte.**

Un sistema está compuesto de partes que interactúan. El comportamiento de un componente específico, como es el software, sólo puede ser definido en el contexto del sistema completo. Por tanto, la especificación debe describir todos los componentes del sistema, estableciendo las interfaces entre estos componentes, y no sólo el componente software

- **La especificación debe abarcar también el entorno del sistema.**

Por el mismo motivo, hay que especificar también el entorno en el que opera el sistema. La especificación del entorno permite describir la interfaz del sistema de la misma forma que las interfaces de los componentes del mismo. De esta forma podemos representar sistemas dinámicos cuyo comportamiento varía dependiendo de los estímulos que reciban del entorno.

Esta especificación del entorno no se hace con vistas a implementarlo, puesto que ya nos viene dado y no podemos modificarlo, sino que sirve para definir los límites del sistema y su interfaz, permitiendo además probarlo.

- **La especificación debe ser operativa.**

La especificación ha de servir para determinar si una implementación concreta la satisface. Esto puede hacerse bien mediante la verificación de la corrección del sistema implementado, cosa que normalmente no puede demostrarse, o bien mediante una serie de casos de prueba elegidos arbitrariamente.

A partir de los resultados de una implementación sobre un conjunto arbitrario de datos de entrada, debe ser posible usar la implementación para validar estos resultados, a pesar de que la especificación no describa el *cómo* sino solamente el *qué*.

Este principio no establece que la especificación tenga que ser ejecutable, sino más bien que pueda ser utilizada para demostrar teoremas.

- **La especificación debe ser ampliable y tolerante a la incompletitud.**

Una especificación es un modelo o abstracción de un sistema real, por tanto nunca será completa, y puede desarrollarse a distintos niveles de detalle. Normalmente

el desarrollo de especificaciones se hace de forma incremental, estando en cada momento elementos del sistema parcialmente especificados. Aunque esto debilita el análisis, las herramientas de prueba de las especificaciones y de comprobación de que las implementaciones son correctas deben ser capaces de manejar especificaciones incompletas.

- **La especificación debe estar localizada y débilmente acoplada.**

Durante su desarrollo, una especificación sufre continuas modificaciones. Por este motivo, la estructura de la especificación debe permitir que estas modificaciones se realicen lo más fácilmente posible. El principio de localidad establece que si es necesario modificar un elemento de la especificación, esta modificación se realice sólo en un punto de la misma. El principio del acoplamiento débil establece que se puedan añadir y quitar partes de la especificación fácilmente.

Para cumplir estos dos principios la especificación ha de ser no redundante y modular, con interfaces breves y bien definidas.

4.6.2.- Características del documento de especificación.

Piattini en cambio nos propone una serie de características que la especificación debe cumplir para ser un documento útil.

1. No ambigua
2. Completa.
3. Fácil de verificar.
4. Consistente.
5. Fácil de modificar.
6. Facilidad para identificar el origen y las consecuencias de cada requisito.
7. Facilidad de utilización durante la fase de explotación y de mantenimiento.

1.- No ambigua

Un requisito ambiguo se presta a distintas interpretaciones. Por lo tanto, un documento de ERS no es ambiguo si y sólo si cada requisito descrito tiene una única interpretación. Esto implica que cada característica del producto final sea descrita utilizando un término único y, además, en los casos en los que un término usado en distintos contextos pueda tener distintos significados, debe incluirse en un **glosario** en el que se determina, de forma específica, su significado.

Con frecuencia, los requisitos se describen en lenguaje natural (por ejemplo, en castellano), lo que implica un gran riesgo, ya que este tipo de lenguaje cuenta con un alto potencial de ambigüedad. Los analistas que especifiquen los requisitos con un lenguaje natural deben poner especial atención en no caer en ambigüedades. Una alternativa que evita estos problemas es el uso de un lenguaje formal de especificación de requisitos

2.- *Completa*

Una ERS está completa si:

1. Incluye todos los requisitos significativos del software, ya sean relativos a la funcionalidad, ejecución, imperativos de diseño, atributos de calidad o a interfaces externas.
2. Define la respuesta del software a todas las posibles clases de datos de entrada y en todas las posibles situaciones. Nótese que es importante el especificar las respuestas tanto para entradas válidas como no válidas.
3. Está conforme con cualquier estándar de especificación que se deba cumplir. La adaptación a una norma puede implicar que si una sección particular del estándar no es aplicable al desarrollo del que se trata, la ERS debe incluir el correspondiente número de sección del estándar y una explicación que justifique su no aplicación.
4. Están etiquetadas y referenciadas en el texto todas las figuras, tablas y diagramas. También deben estar definidos todos los términos y unidades de medida.

Cualquier ERS que utilice la expresión «por determinar» (TBD: To Be Determined) no está completa. Sin embargo, hay veces que suele ser necesario utilizar un TBD y, en este caso, se debe acompañar de:

- Una descripción de las condiciones que han causado el TBD para que la situación pueda resolverse (por ejemplo, porque la administración aún no ha determinado el formato exacto de un impreso de pago de impuestos).
- Una descripción de qué hay que hacer para eliminar el TBD (por ejemplo, esperar a la publicación de un Real Decreto o un reglamento de impuestos).

3.- *Fácil de verificar*

Una ERS es fácil de verificar si y sólo si cualquier requisito al que se haga referencia se puede verificar fácilmente, es decir, si existe algún procedimiento finito y efectivo en coste para que una persona o una máquina compruebe que el software satisface dicho requisito. Un ejemplo de requisito difícil de verificar sería el siguiente:

«El programa no debe entrar nunca en un bucle infinito». La comprobación de este requisito es, incluso teóricamente, imposible. Si no se encuentra un método para determinar si el producto software satisface un requisito concreto, entonces se debe eliminar dicho requisito de la ERS. También puede ocurrir que un requisito no se pueda expresar de forma que se pueda verificar fácilmente cuando se redacta la ERS. Esto no constituye ningún problema si, posteriormente, se puede reescribir en el proyecto en una forma verificable.

4.- *Consistente*

Una ERS es consistente si y sólo si ningún conjunto de requisitos descritos en ella son contradictorios o entran en conflicto. Se pueden presentar tres tipos distintos de conflicto:

1. Dos o más requisitos pueden describir el mismo objeto real pero utilizan distintos términos para designarlo.
2. Las características especificadas de objetos reales pueden estar en conflicto. Por ejemplo, un requisito establece que todas las luces han de ser azules y otro que han de ser verdes.
3. Puede haber un conflicto lógico o temporal entre dos acciones determinadas. Por ejemplo, un requisito puede establecer que se deben sumar dos entradas y otro que han de multiplicarse.

5.- Fácil de modificar

Una ERS es fácilmente modificable si su estructura y estilo permiten que cualquier cambio necesario en los requisitos se pueda realizar fácil, completa y consistentemente. Esto implica que la ERS debe:

- Tener una organización coherente y manejable, con una tabla de contenidos, un índice y referencias cruzadas.
- No ser redundante; o sea, que el mismo requisito no aparezca en más de un lugar de la ERS.

La redundancia en sí no es un error, pero puede fácilmente conducir a errores. Ocasionalmente, la redundancia puede ayudar a la legibilidad de la ERS, pero seguramente provocará problemas cuando haya que actualizar el documento. Por ejemplo, supóngase que un requisito se define en dos lugares distintos de la ERS y posteriormente se decide que dicho requisito debe modificarse. Si el cambio sólo se realiza en uno de los lugares, la ERS quedará inconsistente.

Como la redundancia es difícil de evitar, lo mejor es crear referencias cruzadas entre los requisitos y los términos empleados para definirlos, facilitando así las posibles modificaciones en la ERS.

6.- Facilidad para identificar el origen y las consecuencias de cada requisito.

Se dice que una ERS facilita las referencias con otros productos del ciclo de vida si establece un origen claro para cada uno de los requisitos y si posibilita la referencia de estos requisitos en desarrollos futuros o en incrementos de la documentación. Se recomiendan dos tipos de referencias:

1. Referencias hacia atrás (esto es, a documentos previos al desarrollo). Depende de que los requisitos refieran explícitamente sus fuentes en documentos previos.
2. Referencias hacia adelante (esto es, a los documentos originados a partir de la ERS). Depende de que cada requisito de la ERS tenga un nombre o número de referencia único que sirva para identificarlo en futuros documentos.

Cuando un requisito de la ERS representa un desglose o una derivación de otro requisito, se debe facilitar tanto las referencias hacia atrás como hacia adelante en el ciclo de vida. Las referencias hacia adelante de la ERS son especialmente importantes para el mantenimiento de software. Cuando el código y los documentos son modificados, es esencial poder comprobar el conjunto total de requisitos que pueden verse afectados por estas modificaciones.

7.- Facilidad de utilización durante la fase de explotación y mantenimiento

La ERS también debe tener en cuenta las necesidades de mantenimiento, incluyendo la sustitución eventual del software, especialmente debido a que:

- 1.- El personal que no ha estado relacionado con el desarrollo del producto software se encarga del mantenimiento. Debe recordarse que las pequeñas correcciones suelen afectar sólo al código o al diseño detallado, por lo que se pueden documentar comentando adecuadamente el código y/o el diseño. Sin embargo, cuando las modificaciones son más profundas, es esencial actualizar la documentación del diseño y de los requisitos. Por lo tanto, para este último caso:
 - La ERS debe ser fácilmente modificable, como vimos con anterioridad.
 - La ERS debería prever un registro de las características especiales de cada componente, tales como:
 - Su criticidad (por ejemplo, en los componentes en los que un fallo puede causar mayor daño).
 - Su relación con necesidades temporales (por ejemplo, un listado o una pantalla que puede que no sean necesarios en poco tiempo).
 - Su origen (por ejemplo, la función X o la pantalla Y proceden íntegramente de un producto software existente).
- 2.- Gran parte de los conocimientos y de la información necesaria para el mantenimiento se dan por supuestos en la organización del desarrollo, pero suelen estar ausentes en la organización del mantenimiento. Si no se entiende la razón del origen de una función, es prácticamente imposible desarrollar el mantenimiento.

4.6.3.- Estructura para la ERS

Existen numerosos esquemas para la especificación del software, que varían dependiendo del método de análisis elegido. A título general, podemos proponer el siguiente:

Especificación de requisitos del software.

I. Introducción.

- A. Referencia del sistema.
- B. Descripción general.
 - 1. Objetivos.
 - 2. Restricciones.

II. Descripción de la información.

- A. Representación del flujo de la información.
 - 1. Diagramas de flujo de datos.
 - 2. Diagramas de flujo de control.

- B. Representación del contenido de la información.

III. Descripción funcional.

- A. Narrativa de procesamiento.
- B. Restricciones.

- C. Requisitos de rendimiento.
- D. Restricciones de diseño.

IV. Descripción del comportamiento.

- A. Especificaciones de control.
- B. Estados del sistema.
- C. Eventos y acciones.
- D. Restricciones de diseño.

V. Criterios de validación.

- A. Descripción de las pruebas.
- B. Respuesta esperada del software.
- C. Consideraciones especiales.

VI. Apéndices.

Este esquema se corresponde con un análisis realizado siguiendo métodos estructurados, pero podríamos adaptarlo fácilmente al AOO.

La introducción describe el software en el contexto del sistema del que va a formar parte, indicando cuáles son los objetivos y restricciones del sistema software.

La sección de descripción de la información da una descripción detallada del problema que tiene que resolver el software, indicando los flujos de información que se mueven entre las partes del mismo y el contenido de estos flujos de información (mediante un diccionario de datos). Si usamos análisis estructurado, aquí irían los DFDs y DFCs. Si usamos AOO, incluiríamos aquí los diagramas de objetos.

En la sección de descripción funcional figurarían las definiciones de cada una de las funciones que componen el sistema, mediante la especificación de las primitivas de proceso. Además, hay que definir y justificar todas las restricciones, incluidas las de rendimiento y de diseño que deban satisfacer estas funciones. En el caso del AOO, figurarían aquí también los DFDs.

La sección de descripción del comportamiento debe incluir las especificaciones de control del sistema, normalmente en forma de Diagramas de Estados, definiendo también, si es necesario, las acciones y actividades a realizar y los eventos o sucesos que disparan las transiciones.

Por último, la sección de criterios de validación debe definir las pruebas que permitan determinar si una implementación del sistema satisface la función, restricciones y rendimiento establecidos en la especificación. Esto no es una tarea trivial, pues precisa un buen conocimiento de los requisitos del software y no debemos dejarla para cuando el software ya esté acabado, puesto que entonces se tiende a pasarla por alto o a realizar las pruebas precisamente en función del software construido y no en función del proyectado.

Al igual que la especificación del sistema, la especificación de requisitos del software debe sufrir una revisión realizada conjuntamente por el cliente y el ingeniero del software. Esta revisión puede hacerse a dos niveles. En primer lugar, para determinar si el sistema cumple con la función, restricciones y rendimiento requeridos

por el cliente, y para ver si ha de cambiarse algunas de las estimaciones de coste, riesgo o agenda definidas en el análisis del sistema.

El segundo nivel de revisión es mucho más detallado, y tiene por objeto detectar imprecisiones o ambigüedades en la especificación. Hay que revisar los términos vagos, como ‘a veces’, ‘a menudo’ o ‘alguno’ y las listas no exhaustivas, que acaban en ‘etc.’ o ‘y otros’ e intentar precisarlos. También hay que tener cuidado con las expresiones de certeza, como ‘siempre’, ‘todos’ o ‘nunca’ y comprobar si efectivamente son correctos (ej. el detalle de una factura siempre cabe en una página). En muchos casos, expresiones como ‘obviamente’ o ‘por tanto’ no son tan obvias e intentan ocultar una laguna en la especificación (ej. hoy es miércoles, por tanto, llueve).

4.7.- Validación de requisitos

Esta validación muestra que éstos son los que definen el sistema que el cliente desea. Tiene mucho en común con el análisis, ya que implica encontrar problemas con los requisitos. Sin embargo, son procesos distintos puesto que la validación comprende un bosquejo completo del documento de requisitos mientras que el análisis implica trabajar con requisitos incompletos. Las consecuencias y los costos de los errores en el documento de requisitos ya han sido discutidas sobre los ciclos de vida por lo que no se hará más hincapié en este punto.

Durante el proceso de validación de requisitos, se deben llevar a cabo diferentes tipos de verificación de requisitos en el documento de requisitos. Estas verificaciones incluyen:

1. *Verificaciones de validez*: Un usuario puede pensar que se necesita un sistema para llevar a cabo ciertas funciones. Sin embargo, el razonamiento y el análisis identifican que se requieren funciones adicionales y diferentes. Los sistemas tienen diversos usuarios con diferentes necesidades y cualquier conjunto de requisitos es inevitablemente un compromiso en el entorno del usuario.
2. *Verificaciones de consistencia*: Los requisitos en el documento no deben contradecirse. Esto es, no debe haber restricciones contradictorias o descripciones diferentes de la misma función del sistema.
3. *Verificaciones de integridad*: El documento de requisitos debe incluir requisitos que definan todas las funciones y restricciones propuestas por el usuario del sistema.
4. *Verificaciones de realismo*: Utilizando el conocimiento de la tecnología existente, los requisitos deben verificarse para asegurar que se pueden implementar. Estas verificaciones también deben tomar en cuenta el presupuesto y calendarización del desarrollo del sistema.
5. *Verificabilidad*: Para reducir las discusiones entre el cliente y el contratista, los requisitos del sistema siempre deben redactarse de tal forma que sean verificables. Esto significa que puede diseñarse un conjunto de verificaciones para demostrar que el sistema a entregar cumple esos requisitos.

Existen varias técnicas de validación de requisitos que pueden utilizarse en conjunto o de forma individual:

1. *Revisiones de requisitos* Los requisitos son analizados sistemáticamente por un equipo de revisores. Este proceso se discute en la siguiente sección.
2. *Construcción de prototipos* En este enfoque de validación, se muestra un modelo ejecutable del sistema a los usuarios finales y a los clientes. Éstos pueden hacer experimentos con este modelo para ver si cumple sus necesidades reales.
3. *Generación de casos de prueba* De forma ideal, los requisitos deben poder probarse. Si las pruebas para éstos se consideran como parte del proceso de validación, esto a menudo revela los problemas en los requisitos. Si una prueba es difícil o imposible de diseñar, por lo regular esto significa que los requisitos serán difíciles de implementar y deberían ser considerados nuevamente.
4. *Análisis de consistencia automático* Si los requisitos se expresan como un modelo del sistema en una notación estructurada o formal, entonces las herramientas CASE deben verificar la consistencia del modelo.

Las dificultades en la validación de requisitos no deben menospreciarse. Es difícil demostrar que un conjunto de requisitos cumple las necesidades del usuario. Los usuarios deben visualizar el sistema en operación e imaginarse la manera en que éste encajará en su trabajo. Para los profesionales de la computación es difícil llevar a cabo este tipo de análisis abstracto, pero para los usuarios del sistema es aún más difícil. Como resultado, la validación de requisitos claramente descubre todos los problemas en éstos por lo que es inevitable hacer cambios para corregir las omisiones y las malas interpretaciones después de que el documento de requisitos se ha aprobado.

4.7.1.- Revisión de requisitos

Éste es un proceso manual que involucra a varios lectores que verifican el documento de requisitos, tanto del personal del cliente como del contratista, en cuanto a anomalías y omisiones. Las revisiones de requisitos pueden ser informales o formales.

Las revisiones informales sencillamente implican que los contratistas deben discutir los requisitos con tantos stakeholders del sistema como sea posible. Antes de llevar a cabo una reunión para una revisión formal, se pueden detectar muchos problemas y errores en los requisitos simplemente hablando del sistema con los stakeholders.

En la revisión formal de requisitos, el equipo de desarrollo debe "conducir" al cliente a través de los requisitos del sistema, explicándole las implicaciones de cada requisito. El equipo de revisión debe verificar la consistencia de cada requisito y la integridad de estos como un todo. Los revisores también comprueban:

1. *Verificabilidad*. ¿El requisito puede probarse en la realidad?
2. *Comprendibilidad* ¿Los proveedores o usuario finales del sistema comprenden del todo el requisito?
3. *Rastreabilidad* ¿El origen de los requisitos está claramente establecido? Se tiene que ir hacia la fuente del requisito para evaluar el impacto del cambio. El rastreo es importante ya que permite evaluar el impacto del cambio en el resto del sistema. Esto se discute con mayor detalle en la siguiente sección.

4. *Adaptabilidad* ¿El requisito es adaptable: Es decir. ¿el requisito puede cambiarse sin causar efectos de gran escala en los otros requisitos del sistema"

Los conflictos, contradicciones, errores y omisiones en los requisitos deben señalarse durante la revisión y registrarse formalmente. Queda en los usuarios, el proveedor y el desarrollador del sistema negociar una solución para estos problemas identificados.

4.8.- Administración de requisitos

La ERS normalmente necesitará ser cambiada a medida que progresá el producto software. Es casi imposible especificar algunos detalles en el momento en el que se inicia el proyecto; por ejemplo, puede ser imposible definir durante la fase de requisitos todos los formatos de pantalla para un programa interactivo de forma que se garantice que no se modificarán más adelante. Además, es casi seguro que se realizarán cambios adicionales como consecuencia de haber encontrado deficiencias, defectos e inexactitudes que se descubren a medida que el producto evoluciona. Dos consideraciones a tener en cuenta en este proceso son:

1. El requisito debe ser especificado de la forma más completa posible, aun en el caso en que se prevean de forma inevitable revisiones en el proceso de desarrollo. Por ejemplo, los formatos de pantalla deseados deben especificarse lo mejor posible en la ERS, de forma que sirvan de base para el diseño posterior.
2. Debe iniciarse un proceso formal de cambio para identificar, controlar, seguir e informar de cambios proyectados tan pronto como sean identificados. Los cambios aprobados en los requisitos deben ser incluidos en la ERS de forma que permita:
 - 2.1. Suministrar una revisión precisa y completa del rastro de las modificaciones.
 - 2.2. Permitir un examen de fragmentos actuales y reemplazados de la ERS.

La administración de requisitos es el proceso de comprender y controlar los cambios en los requisitos del sistema. El proceso de administración de requisitos se lleva a cabo junto con los otros procesos de ingeniería de requisitos. La planificación comienza al mismo tiempo que la obtención de requisitos inicial y la administración activa de requisitos debe iniciar tan pronto como esté lista la primera versión del documento de requisitos.

4.8.1.- Requisitos duraderos y volátiles

Desde una perspectiva evolutiva, los requisitos son de dos clases:

1. *Requisitos duraderos* Éstos son relativamente estables que se derivan de la actividad principal de la organización y que están relacionados directamente con el dominio del sistema. Por ejemplo, en un hospital siempre habrá requisitos que se refieren a los pacientes, doctores, enfermeras, tratamientos, etcétera.
2. *Requisitos volátiles* Estos cambiarán probablemente durante el desarrollo del sistema o después de que éste se haya puesto en operación. Por ejemplo, por cambios de las políticas gubernamentales de salud.

Sommerville nos ofrece la siguiente división de los requisitos volátiles.

Requisitos mutantes	Requisitos que cambian debido a los cambios en el ambiente en el que opera la organización. Por ejemplo, en los sistemas hospitalarios, la consolidación del cuidado del paciente puede cambiar y requerir un tratamiento diferente de la información a recolectar.
Requisitos emergentes	Requisitos que emergen al incrementarse la comprensión del cliente en el desarrollo del sistema. El proceso de diseño puede revelar requisitos emergentes nuevos.
Requisitos consecutivos	Requisitos que son resultado de la introducción del sistema de cómputo. Esta introducción puede cambiar los procesos de la organización y abrir nuevas formas de trabajar que generarán nuevos requisitos del sistema.
Requisitos de compatibilidad	Requisitos que dependen de sistemas particulares o procesos de negocios dentro de la organización. Cuando estos últimos cambian, los requisitos de compatibilidad del sistema contratado o a entregar también pueden cambiar.

4.8.2.- Planificación de la administración de requisitos.

Ésta es una primera etapa esencial del progreso de administración de requisitos. La administración de requisitos es muy cara y, para cada proyecto, la etapa de planificación establece el nivel de detalle necesario en la administración de requisitos. Durante la etapa de administración de requisitos se tiene que decidir sobre:

1. *La identificación de requisitos* Cada requisito se debe identificar de forma única de tal forma que puedan entrar en referencia cruzada con otros requisitos de manera que pueda utilizarse en las evaluaciones de rastreo.
2. *Un proceso de administración del cambio* Éste es el conjunto de actividades que evalúa el impacto y costo de los cambios.
3. *Políticas de rastreo* Éstas definen la relación entre requisitos y la de éstos y el diseño del sistema que se debe registrar y la manera en que estos registros se deben mantener.
4. *Ayuda de herramientas CASE* La administración de requisitos comprende el procesamiento de grandes cantidades de información de los requisitos. Las herramientas que se pueden utilizar van desde sistemas de administración de requisitos especiales hasta hojas de cálculo y sistemas sencillos de bases de datos.

Existen relaciones entre los requisitos mismos y entre éstos y el diseño del sistema. También existen vínculos entre los requisitos y las razones de por qué éstos se propusieron. Cuando se proponen cambios se tiene que rastrear el impacto de estos cambios en los otros requisitos y el diseño del sistema. El rastreo es una propiedad de la especificación de requisitos que refleja la facilidad de encontrar requisitos relacionados.

Existen tres tipos de información de rastreo a las que se les debe dar mantenimiento:

1. *La información de rastreo de la fuente* vincula los requisitos con los *stakeholders* que propusieron los requisitos y la razón de éstos. Cuando un cambio es propuesto, esta información se utiliza para descubrir a los *stakeholders*, de forma que se les pueda consultar acerca de ese cambio.

2. *La información de rastreo de los requisitos* vincula los requisitos dependientes en el documento de requisitos. Esta información se utiliza para evaluar cómo muchos requisitos se ven probablemente afectados por un cambio propuesto y la magnitud de los cambios consecuentes en los requisitos.
3. *Información de rastreo del diseño* vincula los requisitos a los módulos del diseño en los cuales serán implementados. Esta información se utiliza para evaluar el impacto de los cambios de los requisitos propuestos en el diseño e implementación del sistema.

A menudo, la formación de rastreo implica utilizar matrices de rastreo que relacionan los requisitos con los *stakeholders*, entre los requisitos o entre los módulos del diseño. Si se consideran matrices de rastreo que vinculan requisitos con otros requisitos, cada uno de éstos se representa por una fila y una columna en la matriz. Donde existe una dependencia entre los requisitos ésta se registra en una celda en la intersección fila/columna.

Esto se ilustra en la figura que muestra una matriz de rastreo sencilla en la cual se registran las dependencias entre los requisitos. Una "U" en la intersección fila/columna ilustra que el requisito en la fila utiliza los recursos especificados en el requisito señalado en la columna: una "R" significa que existe una relación débil entre los requisitos. Por ejemplo, pueden definirse requisitos para partes del mismo subsistema.

Req. Id.	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

Las matrices de rastreo se utilizan cuando se tiene que administrar un número pequeño de requisitos pero son muy pesadas y caras de mantener para sistemas grandes con muchos requisitos. Para estos sistemas, se tiene que capturar la información de rastreo en una base de datos de requisitos en la que cada requisito esté explícitamente vinculado a los requisitos relacionados. El impacto de los cambios se puede evaluar utilizando las facilidades de exploración de la base de datos. Alternativamente, es posible generar matrices de rastreo en forma automática.

La administración de requisitos necesita ayuda automática y las herramientas CASE a utilizarse deben elegirse durante la fase de planificación. Se requiere ayuda de las herramientas para:

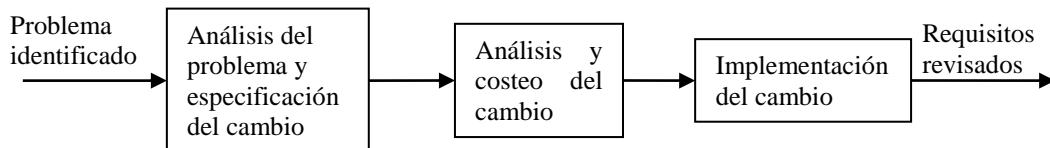
1. *Almacenar requisitos* Los requisitos deben mantenerse en un almacén de datos seguro y administrado que sea accesible a todos los que estén relacionados en el proceso de ingeniería de requisitos.
2. *Administrar el cambio* Este proceso se simplifica si está disponible una herramienta de ayuda.

3. *Administrar el rastreo* Como se discutió anteriormente, las herramientas de ayuda para el rastreo permiten relacionar los requisitos descubiertos. Para ayudar a descubrir las posibles relaciones entre los requisitos están disponibles algunas herramientas que utilizan técnicas de procesamiento de lenguaje natural.

Para sistemas pequeños, no es necesario utilizar herramientas de administración de requisitos especializadas. El proceso de administración de requisitos puede llevarse a cabo utilizando los recursos disponibles en los procesadores de texto, hojas de cálculo y bases de datos en PC. Sin embargo, para sistemas grandes, se requieren herramientas de ayuda más especializadas como DOORS, Requisite Pro o REM.

4.8.3.- Administración del cambio de requisitos.

Esta administración se aplica a todos los cambios propuestos en los requisitos. La ventaja de utilizar un proceso formal para administrar el cambio es que todos los cambios propuestos son tratados de forma consistente y que los cambios en el documento de requisitos se hacen de forma controlada.



Existen tres etapas principales en un proceso de administración de cambio:

1. *Análisis del problema y especificación del cambio* El proceso inicia con un problema de requisitos identificado o, algunas veces, con una propuesta de cambio específica. Durante esta etapa, el problema o la propuesta de cambio se analiza para verificar que ésta es válida. Entonces se hace una propuesta de cambio de requisitos más específica.
2. *Análisis del cambio y costeo* El efecto de un cambio propuesto se valora utilizando la información de rastreo y el conocimiento general de los requisitos del sistema. El costo de hacer un cambio se estima tanto en términos de modificaciones al documento de requisitos y, si es apropiado, como en el diseño e implementación del sistema. Una vez que el análisis se completa, se toma una decisión sobre si se procede o no en el cambio de requisitos.
3. *Implementación del cambio* Se modifica el documento de requisitos y, en su caso, el diseño e implementación del sistema. El documento se organiza para que los cambios puedan acomodarse sin tener que redactarlo todo nuevamente. Como con los programas de eventos, los cambios en los documentos se llevan a cabo minimizando las referencias externas y haciendo que las secciones del documento sean tan modulares como sea posible.

Si se requiere de forma urgente un cambio en los requisitos del sistema, existe siempre la tentación de hacer ese cambio al sistema y entonces modificar de forma retrospectiva el documento de requisitos. Esto inevitablemente conduce a que la especificación de requisitos y la implementación del sistema se desfasen. Una vez que se han hecho los cambios en el sistema, los del documento de requisitos se olvidan o se hacen de forma no consistente con los cambios del sistema.

Tema 5.- Análisis estructurado

Tema 5.- Análisis estructurado	115
5.1.- Introducción.....	116
5.2.- Técnicas de especificación y modelado.....	117
5.2.1.- Diagramas de flujo de datos.	120
5.2.2.- Especificaciones de proceso	124
5.2.3.- Diagramas de flujo de control.....	125
5.2.4.- Especificaciones de control	129
5.2.5.- Diagramas de estados.....	132
5.2.6.- Redes de Petri	136
5.2.7.- Diagramas Entidad/Relación.	138
5.2.8.- Diccionario de datos.	138
5.2.9.- Comprobaciones a realizar sobre una especificación estructurada.....	140
5.3.- Consistencia entre modelos.	141
5.3.1.- Técnicas matriciales.....	144
5.4.- Metodología del análisis estructurado.	145
5.4.1.- Fases.	145
5.5.- Modelos del sistema: esencial y de implementación.	148
5.6.- Ejemplos.	150
5.6.1.- Traductor y Distribuidor de Comunicaciones.....	150
5.6.2.- Hogar Seguro	156

5.1.- Introducción.

Todos los métodos de análisis de requisitos se basan en la construcción de modelos del sistema que se pretende desarrollar. Utilizando alguna notación, propia de cada método, creamos modelos que reflejen el sistema, y aplicamos técnicas de descomposición y razonamiento *top-down*, de tal forma que al final establecemos la esencia del sistema que pretendemos construir.

El desarrollo de modelos presenta algunas ventajas claras¹:

Permite centrarse en determinadas características del sistema, dejando de lado otras menos significativas. Esto nos permite centrar las discusiones con el usuario en los aspectos más importantes del sistema, sin distraernos en características del sistema que sean irrelevantes.

Permite realizar cambios y correcciones en los requisitos a bajo coste y sin correr ningún riesgo. Si nos damos cuenta que no habíamos entendido las necesidades del usuario o si el usuario ha cambiado de idea acerca de los requisitos del sistema, podemos cambiar el modelo o incluso desecharlo y empezar de nuevo. Si no hiciésemos modelos, los cambios en los requisitos sólo se efectuarían después de construir el producto software, y el coste sería muchísimo mayor.

Permite verificar que el ingeniero del software ha entendido correctamente las necesidades del usuario y que las ha documentado de tal forma que los diseñadores y programadores pueden construir el software.

Sin embargo, no todas las técnicas de análisis logran estos tres objetivos: una descripción del sistema de 500 páginas (que sería, en sentido estricto, un modelo) oculta las características del sistema, tanto las relevantes como las irrelevantes, su desarrollo puede costar más que el del propio sistema, es difícil de modificar y no permite verificar si se han establecido los requisitos.

El análisis de requisitos clásico, usado hasta finales de los 70, consistía en redactar especificaciones funcionales, en forma de documentos textuales, de este tipo:

Eran monolíticas. Para entender el sistema había que leerse la especificación de principio a fin. No había posibilidad de que ni el analista ni el usuario pudiesen centrarse en una determinada parte de la especificación, sin tener que leerse el resto.

Eran redundantes. La misma información se repartía en diferentes partes del documento. Debido a esto cualquier cambio que se hiciera en la especificación debía reflejarse en varios puntos del documento. Esta situación produce con frecuencia inconsistencia: si no se cambiaba en todos estos lugares la misma información (p.ej. el mismo código) tenía definiciones distintas.

¹ Pressman5, pag. 200

Eran ambiguas. Al estar escritas en lenguaje natural, podían ser interpretadas de forma distinta por analistas, usuarios, diseñadores o programadores. Hay estudios que muestran que el 50% de los errores encontrados en el sistema final y el 70% del coste de la corrección de los errores surgía de este tipo de malentendidos.

Eran imposibles de mantener o modificar. Por todas las razones anteriores, la especificación del sistema estaba totalmente obsoleta cuando finalizaba el desarrollo. En muchos casos, estaba incluso obsoleta cuando finalizaba la fase de análisis de requisitos. Debido a esto, la mayor parte del software de la época carece de una documentación fiable, incluso aunque se hubiese realizado análisis y redactado una especificación.

Por estos motivos fueron surgiendo nuevos métodos de análisis, cuyo objetivo era obtener especificaciones:

- **Gráficas.** Formadas por una colección de diagramas, acompañados de información textual detallada, que sirve de material de referencia, más que de cuerpo principal de la especificación.
- **Particionadas.** De forma que fuese posible leerse o trabajar sobre partes individuales de la especificación sin tener que leérsela toda.
- **Mínimamente redundantes.** De forma que los cambios en los requisitos necesitasen reflejarse en un sólo punto de la especificación.
- **Transparentes.** De forma que fuesen tan fáciles de leer y de entender que el que las utilizase no se diese cuenta de que está mirando una representación del sistema, en lugar del sistema en sí. Los sistemas son los que son complejos, las especificaciones tienen que ser claras y sencillas.

En este tema nos centraremos en los *Métodos de análisis estructurado*, que son los más utilizados en la actualidad. En los temas siguientes veremos métodos de análisis orientado a objetos y utilizando lenguajes de especificación formal.

5.2.- Técnicas de especificación y modelado

Bajo el nombre genérico de *Análisis estructurado*, se engloban una serie de aportaciones de diversos autores entre los que cabe citar, por orden cronológico, a De Marco, Yourdon, Gane & Sarson, Ward & Mellor y Hatley & Pirbhai. Cada uno de ellos ha desarrollado su propio método de análisis, mejorando, ampliando o adaptando los anteriores a algún campo de aplicación específico.

Siguiendo las técnicas del análisis estructurado podemos describir los sistemas desde tres puntos de vista:

Punto de vista de los datos. Dimensión de la información

Se centra en la información que utiliza el sistema. Representaremos el modelo de los datos que utiliza el sistema, haciendo explícitas las relaciones que se establecen entre esos datos. Para ello utilizaremos **Diagramas Entidad/Relación**.

El desarrollo del modelo de datos de un sistema evita el almacenamiento de información redundante e incoherente, garantiza la integridad y seguridad de los datos y simplifica el mantenimiento.

Punto de vista del proceso. Dimensión de la función

Se centra en qué hace el sistema. Para ello lo describiremos como un conjunto de operaciones de proceso de información. Estas operaciones reciben unos flujos de datos de entrada y los transforman en flujos de datos de salida. Para describir el sistema desde este punto de vista utilizaremos **Diagramas de Flujo de Datos** y **Especificaciones de procesos**.

Punto de vista del comportamiento. Dimensión del tiempo

Se centra en cuándo suceda algo en el sistema. Describiremos el sistema como una sucesión de estados o modos de funcionamiento. Indicaremos también cuáles son las condiciones o eventos que hacen que el sistema pase de un modo a otro. Utilizaremos **Diagramas de Flujo de Control**, **Especificaciones de Control** y **Diagramas de Estados**.

Cada sistema tiene una representación más o menos significativa en cada una de estas dimensiones y por tanto para cada sistema adquirirán más o menos importancia las técnicas que se centran en cada dimensión. Por ejemplo, un sistema de información basado en una gran Base de Datos tendrá una importantísima componente en la dimensión de la información, y por tanto, su representación se centrará en las técnicas que permiten modelar esta dimensión. Un sistema de gestión en tiempo real en cambio, como por ejemplo la centralita electrónica de encendido de un motor de combustión, tendrá el tiempo como componente más importante.

Aunque se han presentado dos casos extremos en los que una dimensión predomina sobre las demás hasta el punto en el que la representación en las otras podría ser obviada lo normal, en general, la mayoría de los sistemas requerirán utilizar modelos en varias dimensiones para representarlos. Cada modelo se centra en un número limitado de aspectos del sistema, dejando de lado otros (ésta era una de las características de los modelos). Combinando los diversos modelos podemos tener una visión detallada de todas las características del sistema. Esto es especialmente cierto hoy en día, cuando los sistemas software manejan estructuras de datos complejas, realizan funciones complejas y tienen pautas de comportamiento complejas.

Es importante dejar claro, sin embargo, que todos los modelos describen un único sistema y por tanto es razonable pensar en estrategias que nos permitan relacionar las distintas representaciones, y por tanto, la consistencia entre los modelos. Dichas

técnicas se encuentran en los planos formados por la combinación de dos dimensiones, por ejemplo, el plano Información-Función.

La tabla siguiente representa una posible clasificación de las técnicas según su dimensión. Las que tengan la misma fila y columna se centrarán en una dimensión mientras que las otras harán referencia al plano formado por las dimensiones indicadas por su fila y su columna.

	Información	Función	Tiempo
Información	Diagramas Entidad-Relación (ER).		
Función	Diagramas de Flujo de datos (DFD)	Diagramas de Flujo de datos (DFD).	
Tiempo	Diagramas de Historia y vida de entidad.	Redes de Petri Diagramas de transición de estados	Diagramas de transición de estados Diagramas de flujo de control

Fig. 5.1. Diferentes técnicas de modelado.

Dichas técnicas buscan modelar a alto nivel el sistema en cada una de sus dimensiones. Las técnicas siguientes dan el máximo nivel de detalle posible de aquel aspecto que representan y por tanto se presentan como técnicas de especificación.

	Información	Función	Tiempo
Información	Especificación de entidad		
Función		Diccionario de datos Especificación de procesos Especificación de entidades externas	
Tiempo		Definición de función	Especificación de eventos

Fig. 5.2. Diferentes técnicas de especificación.

Indicar únicamente que el **Diccionario de Datos** puede incluir, según el autor, la especificación de entidad, procesos, entidades externas, eventos y naturalmente la de los datos con lo que nos permite definir y relacionar todos los elementos presentes en las distintas representaciones.

En este punto vamos a estudiar las diferentes técnicas y notaciones de modelado de sistemas que puede utilizar un ingeniero del software. Combinando todas ellas se puede establecer un modelo completo del sistema, pero lo importante es usar aquellos diagramas que nos sirvan en una situación determinada.

Por último, hay que tener en cuenta la influencia que han tenido las herramientas de análisis en el uso y difusión de los métodos de análisis estructurado. Estas herramientas permiten dibujar los diagramas, hacen mucho más sencilla su modificación y comprueban su completitud y consistencia. Además muchas de estas herramientas sirven de soporte no sólo a la fase de análisis sino a todas las etapas del ciclo de vida. Estas son las herramientas CASE.

5.2.1.- Diagramas de flujo de datos.

Como su propio nombre indica, un sistema de procesamiento de datos incluye tanto datos como procesos, y cualquier análisis de un sistema así debe incluir ambos aspectos. Necesitamos una técnica para modelar sistemas que describa:

- ◆ Qué funciones son las que realiza el sistema.
- ◆ Qué interacción se produce entre estas funciones.
- ◆ Qué transformaciones de datos realiza el sistema.
- ◆ Qué datos de entrada se transforman en qué datos de salida.

A medida que la información se mueve a través del software, va siendo modificada mediante una serie de transformaciones. El DFD es una técnica gráfica que utiliza un diagrama en forma de red para representar el flujo de información y las transformaciones que se aplican a los datos al moverse desde la entrada a la salida.

Elementos de un DFD.

Para representar el sistema mediante DFDs utilizaremos la notación de Yourdon (1975), posiblemente la más extendida. Esta notación es la indicada en la transparencia 4.3. Los elementos que aparecen en el DFD pueden ser:

Procesos. Representan elementos software que transforman información. Son, por tanto, los componentes software que realizan cada una de las funciones del sistema, transformando datos de entrada en datos de salida. Los representaremos como cuadrados de esquinas redondeadas, que llevan asociado un número y un nombre de proceso.

- ◆ *Regla de conservación de datos:* El proceso debe ser capaz de generar las salidas a partir de los flujos de entrada más una información local
- ◆ *Pérdida de información:* Una entrada no se utiliza en un proceso para generar ningún flujo de salida

Entidades externas. Representan elementos del sistema informático o de otros sistemas adyacentes (en cualquier caso se trata de algo que está fuera de los límites del sistema software) que producen información que va a ser transformada por el software o que consumen información transformada por el software. Los flujos de datos que comuniquen el sistema con las entidades externas representan las interfaces del sistema. Los flujos entre unidades externas no son objeto de estudio y nunca deben representarse, si son necesarios hay que replantearse los límites del software. Las entidades externas sólo aparecen en el diagrama de contexto. Serán representados por un cuadrado con el nombre identificativo de la entidad externa.

Almacenes de datos. Representan información almacenada que puede ser utilizada por el software. Los almacenes de datos permiten guardar temporalmente información que luego puede ser procesada por el mismo proceso que la creó o por otro distinto. En la mayoría de los casos, utilizaremos almacenes de datos cuando dos procesos intercambian información pero no están sincronizados, esto es, el proceso destino no comienza a procesar la información en cuanto le llega. En otros casos,

utilizaremos los almacenes como copia de seguridad de los datos, para evitar pérdidas de información en caso de que el sistema falle. Los almacenes de datos pueden ir desde registros temporales para almacenar un dato hasta ficheros o bases de datos. En nuestro caso los representaremos por dos barras paralelas unidas por sendos arcos.

Flujos de datos. Representan datos o colecciones de datos que fluyen a través del sistema. La flecha indica el sentido de flujo. Posiblemente en los diagramas de nivel mayor existan flujos de datos bidireccionales (par de diálogo), que luego son refinados en sucesivos diagramas, o incluso varios flujos de datos agrupados en uno sólo (flujos múltiples). Los flujos de datos conectan los procesos con otros procesos, con entidades externas o con almacenes de datos, y pueden converger o divergir si conectan un elemento del DFD con varios otros. Mientras que los almacenes de datos representan información estática o en reposo, los flujos de datos representan información en movimiento. Puede tratarse de un elemento de datos simple o compuesto (un registro) o incluso de una colección de datos de estructura compleja, p.ej. un árbol. En algunos casos, el flujo de datos puede representar elementos que no son datos, sino p.ej. materiales, si estamos haciendo un diagrama de flujo de una cadena de producción, por ejemplo. Los flujos de datos se representan por flechas que llevan asociado un nombre.

- ◆ Por tanto, los flujos contienen información de las tres dimensiones aunque normalmente sólo se representan la de Función e Información:
 - Según su *dimensión temporal* los flujos pueden ser:
 - Discretos: Si representan movimientos de datos en un instante determinado del tiempo. →
 - Continuos: Si implican una transición continua de información, por ejemplo para comprobar si un registro ha cambiado. →→
 - Según la *dimensión de la Función* los flujos pueden ser:
 - De consulta: Salen de un almacén. Utilizan la información del almacén pero no la modifican.
 - De actualización: Entran en un almacén. Crean, modifican o borran datos del Almacen.
 - De Dialogo: Representa simultáneamente un proceso de Consulta y otro de actualización. Flujo de doble flecha y con dos nombres, uno hace referencia al iniciador y el otro es la respuesta. Por ejemplo un gestor de almacén de piezas mecánicas podría comprobar si hay un tipo de pieza y si la hay indicar que queda una menos; ambas acciones quedarían reflejadas en un flujo de diálogo.
 - Finalmente en la *dimensión de la Información* los flujos pueden ser de varios tipos atendiendo a su contenido que puede ser:
 - Un elemento: Contiene un dato elemental o una pieza indivisible de información.
 - Un Grupo: Contiene varios datos elementales
 - Múltiples: En el DFD se representa como un único flujo pero en realidad está formado por un conjunto de ellos.

Cualquiera de los elementos que aparecen en un DFD tiene que estar etiquetado con un nombre, corto y significativo que debe ser único en el conjunto de DFDs que representan al sistema. Los procesos van etiquetados con la función que realizan. Lo mismo sucede con las entidades externas, que también se pueden etiquetar con el

nombre de la máquina, persona o grupo de personas que realizan esa función, en el caso de que no se trate de software. Los flujos de datos van etiquetados con un nombre identificativo de la información que transportan, y posiblemente con el estado de dicha información (p. ej. *número de teléfono*, *número de teléfono correcto*, *número de teléfono erróneo*). Los almacenes de datos van etiquetados con un nombre significativo de la información que contienen, generalmente en plural. Es importante destacar que los nombre deben ser representativos lo que quiere decir que deben hacer referencia a toda la función o información de aquello que identifican y no solamente a una parte y que deben evitarse que sean poco significativos tales como “Realizar operación” o “gestionar acción”.

Hay que tener en cuenta que un DFD no representa información sobre el comportamiento del sistema o sobre el control del mismo. Representa qué funciones o qué transformaciones se realizan sobre los datos pero no cuándo se realizan o en qué secuencia.

Diagrama de contexto.

Se pueden utilizar DFDs para representar el sistema a cualquier nivel de abstracción. El DFD de nivel 0 se llama **diagrama de contexto** y en él el sistema está representado por un sólo *proceso*, que identifica cuál es la función principal del sistema, mostrando además los flujos de información que lo relacionan con otros sistemas: las *entidades externas*. El diagrama de contexto tiene una gran importancia puesto que resume el requisito principal del sistema de recibir ciertas entradas, procesarlas de acuerdo con determinada función y generar ciertas salidas. A partir del diagrama de contexto podemos ir construyendo nuevos diagramas que vayan definiendo con mayor nivel de detalle los flujos de datos y procesos de transformación que ocurren en el sistema, de forma que al final obtenemos una jerarquía de diagramas.

En general, cualquier proceso que aparezca en un DFD puede ser descrito más detalladamente en un nuevo DFD. A esto lo llamaremos **explosión** de un proceso. En este DFD el proceso que estamos describiendo aparece descompuesto en una serie de subprocessos o subsistemas, cada uno encargado de realizar un aspecto determinado del proceso original. Los flujos de datos que entran y salen del proceso que estamos describiendo deben entrar y salir del DFD que lo desarrolla. Además de estos flujos, el DFD contendrá por lo general nuevos flujos que comunican los procesos que figuran en él y posiblemente almacenes de datos. Las entidades externas sólo aparecen en el DFD de contexto.

Dado que en el DFD de contexto nuestro sistema estará representado por un sólo proceso. Este DFD de nivel 0 sólo dará lugar a un DFD de nivel 1. A su vez, este puede dar lugar a tantos DFDs de nivel 2 como procesos contenga, y así sucesivamente hasta que hayamos alcanzado un nivel en el que los procesos sean lo suficientemente simples como para no necesitar su descripción más detallada en un DFD. De este forma, el modelo de procesos del sistema va a consistir en una jerarquía de DFDs.

Una ventaja de los DFDs es que no sólo se utilizan para representar modelos del software sino también en muchos otros campos, como la investigación operativa o la

organización empresarial. Eso hace que estén ampliamente difundidos y que sea relativamente fácil mostrarlos al usuario y que éste los entienda.

DFDs. Reglas de construcción.

- ◆ *Un DFD debe contener menos de 10 elementos.*
- ◆ *Cada elemento de un DFD debe tener un nombre corto e identificativo.*
- ◆ *Es necesario numerar los procesos.*
- ◆ *Para modelar sistemas complejos se utiliza la explosión, que da como resultado DFDs a distintos niveles de detalle.*
- ◆ *No es conveniente utilizar más de 7 u 8 niveles.*
- ◆ *Los DFDs de niveles inferiores desarrollan de forma más concreta los procesos de niveles superiores.*
- ◆ *La explosión se realiza hasta alcanzar un nivel de especificación mínimo y sencillo.*
- ◆ *Debe mantenerse la consistencia de nombres en los distintos DFDs.*
- ◆ *Debe mantenerse la consistencia entre los distintos niveles, utilizando la regla de balanceo.*
- ◆ *En cada DFD hijo deben representarse los mismos flujos de datos que en el proceso padre.*
- ◆ *No existen conexiones entre entidades externas*
- ◆ *No existen conexiones entre entidades externas y almacenes*
- ◆ *No existen conexiones entre almacenes*

Regla de balanceo.

Normalmente, el sistema que estamos modelando será lo suficientemente complejo como para necesitar varios DFDs. Construiremos entonces una jerarquía de DFDs.

Cada DFD (hijo) de un nivel n será resultado de la explosión de un proceso (padre) de un DFD de nivel $n-1$. Es necesario que el título del DFD sea el nombre del proceso que desarrolla, que la numeración de los procesos en el DFD hijo se derive del número del DFD padre (p.ej. 3, 3.1, 3.2) y además hay que mantener la consistencia de los flujos de datos en ambos.

La regla de balanceo dice que los flujos de datos que entran o salen del proceso padre deben aparecer en el DFD hijo, manteniendo el nombre y el sentido.

En el diagrama hijo estos flujos de datos tendrán un extremo libre, puesto que conectaban el proceso padre con algún elemento que ya no va a estar representado en el diagrama hijo. Una excepción son los flujos que conectan los procesos con almacenes de datos. Se recomienda representar los almacenes de datos en todos los DFDs en los que intervienen.

La regla de balanceo también puede tener excepciones. Antes habíamos hablado de flujos de datos compuestos, e incluso bidireccionales (que normalmente transmiten

una pregunta o petición y su respuesta). Estos flujos de datos suelen utilizarse en los DFDs de los niveles más bajos (es decir, los más altos de la jerarquía) para modelar el sistema con cierto grado de abstracción y para evitar representar muchos flujos de datos, lo que haría más confuso el diagrama. En los DFDs de niveles superiores, es necesario descomponer estos flujos en varios (p.ej. petición y respuesta). Para realizar esta descomposición podemos hacer divergir el flujo en el DFD hijo o al menos dejar reflejada esta descomposición en el diccionario de datos (que ya se verá qué es).

Ejemplos:

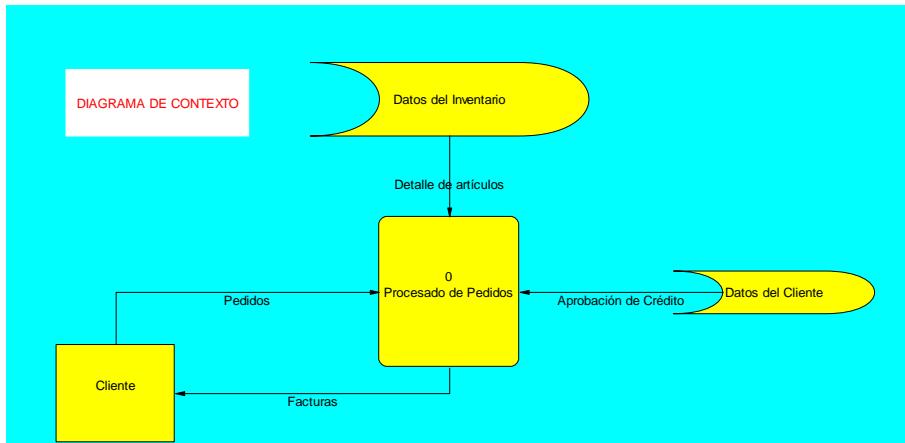


Fig. 5.3. Diagrama de contexto de un software de procesado de pedidos

Imaginémonos que tenemos que desarrollar un software para gestionar los pedidos que recibe una determinada empresa. El diagrama de contexto será el que tenemos en la figura 5.2. Simbolizamos todo el sistema como un solo proceso (cuadrado con esquinas redondeadas) que interactúa con el resto del “mundo”. Este mundo o entorno estaría formado por el cliente (cuadrado con esquinas, ya que es un agente externo) que efectúa pedidos a nuestro software y recibe facturas de él (dos flujos de datos con sentidos diferentes), por la base de datos del inventario (externa a nuestro software) contra la que confirmaremos la existencia de unidades suficientes para satisfacer el pedido del cliente, y la base de datos fiscales de nuestros clientes de la que nuestro software extraerá información para generar la factura correspondiente.

5.2.2.- Especificaciones de proceso

Los DFDs permiten identificar las principales funciones o transformaciones que realiza un sistema, y las relaciones entre estas funciones pero no indican nada acerca de los detalles de cómo se realizan estas transformaciones. Para definir los detalles de qué información de entrada se transforma en qué información de salida y cómo se realiza esta información se necesita una descripción textual de los procesos. Para esto utilizaremos las especificaciones de proceso.

En los DFDs de menor nivel (los más altos en la jerarquía), los procesos se describen mediante un nuevo DFD, que define más detalladamente las funciones que realiza y los flujos que maneja. Este proceso de descomposición debe continuar hasta

que se alcance un nivel en el que un proceso puede ser descrito textualmente de forma sencilla y no ambigua. Estos procesos de los nodos hoja de la jerarquía se suelen llamar **primitivas de proceso**.

Una forma de especificar una primitiva de proceso es dando un algoritmo. Esto no quiere decir que el algoritmo propuesto en el análisis se corresponda con la implementación final del proceso, sino que es simplemente una forma de describir la operación. La elección del algoritmo definitivo se hace en la fase de diseño, considerando además de la especificación, criterios de eficiencia y las características del lenguaje de implementación.

Alternativamente, una primitiva de proceso puede ser descrita mediante una definición, sin especificar un algoritmo (p.ej. el proceso *Invertir Matriz* calcula la inversa de la matriz que recibe como entrada), matemáticamente o mediante un lenguaje de especificación formal, en lenguaje natural o mediante una tabla que indica los valores de los flujos de salida a partir de los valores de los flujos de entrada.

En cualquier caso, las especificaciones de proceso son textos breves (unas pocas líneas, en cualquier caso menos de una página), que definen la función que realiza la primitiva de proceso para transformar los flujos de entrada en flujos de salida.

5.2.3.- Diagramas de flujo de control.

Los DFDs son muy útiles a la hora de modelar sistemas de proceso de datos, describen las funciones o transformaciones que realiza el sistema y cómo fluyen los datos a través del mismo.

Sin embargo, en los DFDs no se representa explícitamente el control o flujo de sucesos del sistema, por lo que estos diagramas no aportan ninguna información sobre cómo se comporta el sistema, sobre cuándo se realizan los procesos o en qué orden. En determinados sistemas de software de gestión, este funcionamiento puede ser intuitivo, y no necesitaremos reflejarlo en el modelo, pero en otras aplicaciones (p. ej. relacionadas con sistemas de tiempo real o con automatismos) sí que es importante que el modelo incluya los aspectos de comportamiento y control del sistema.

Podemos considerar tres situaciones posibles, referidas a cómo se comporta el sistema:

- Los procesos que figuran en el DFD están activos siempre. En este caso no necesitamos especificar el control del sistema.
- Los procesos se activan cuando llegan datos a través de sus flujos de entrada, transforman estos datos y emiten los resultados a través de los flujos de salida, permaneciendo inactivos hasta la llegada de nuevos datos. Este comportamiento está implícito en la notación usada para los DFDs por lo que tampoco será necesario especificar el control. Muchas de las aplicaciones de gestión pueden ser descritas de esta forma.
- Cada proceso pasa por períodos de actividad e inactividad que están gobernados por mecanismos más complejos que los descritos anteriormente. Por lo general, un proceso se activará cuando se produzca determinada situación o suceso en el sistema y permanecerá activo hasta

que se produzca otra situación. Comportamientos de este tipo no pueden ser reflejados de forma adecuada en los DFDs por lo que en estos casos necesitaremos describir el modelo de control o comportamiento del sistema, al menos para estos procesos que siguen patrones de comportamiento complejos. En aquéllas aplicaciones en las que el software controle el funcionamiento de otros dispositivos (p. ej. en un sistema empotrado) nos encontraremos con situaciones de este tipo.

Nota: Hay que tener en cuenta que el modelo de control del sistema, establece el comportamiento de éste a alto nivel: qué procesos se encuentran activos en cada momento o en qué secuencia se realizan las transformaciones de los datos. Existe un control de bajo nivel que se refiere a los saltos condicionales y a los bucles de los programas. Este control de bajo nivel estaría reflejado en las PSPECs de los procesos.

Estas falta de expresividad de los DFDs llevaron a proponer varias extensiones de los mismos para modelar el control del sistema. Las más importantes son las de Ward & Mellor (1985) y Hatley & Pirbhai (1987). Nosotros vamos a seguir la notación de estos últimos.

Para modelar el control del sistema Hatley y Pirbhai proponen el uso de Diagramas de Flujo de Control (DFCs), similares a los DFDs ya vistos. Su método se basa en eliminar de los DFDs todo lo relativo a información de control : sucesos, señales, condiciones de datos, etc. y construir una jerarquía de DFCs paralela a la de DFDs. En estos DFCs se especifica todo el flujo de sucesos, señales y condiciones de datos del sistema, es decir, se indican los elementos de información que intervienen en el control de los procesos.

Nota: Otros autores (W & M) proponen incluir toda la especificación del control en los DFDs, por lo que no utilizan DFCs. Hatley y Pirbhai prefieren separar ambos modelos, para dejar más claro lo que es procesamiento y lo que es control, y para evitar recargar los DFDs con más información. No obstante, en sistemas sencillos con mecanismos de control sencillos sería admisible utilizar un único diagrama combinado, siempre y cuando quede reflejado lo que son datos y lo que son eventos.

Según lo anterior, construiremos una jerarquía de DFCs , empezando por un diagrama de contexto, de forma que a cada DFD (o al menos para aquellos que sea necesario representar su control) le corresponda un DFC. En cada par DFD/DFC representaremos los mismos procesos y las mismas entidades externas, puesto que ambos representan modelos de la misma parte del sistema con el mismo nivel de detalle, aunque con puntos de vista distintos.

Las reglas sobre denominación numeración, relaciones padre - hijo y balanceo que se aplican a los DFCs son las mismas que se establecen para los DFDs.

Los elementos que aparecen en un DFC son prácticamente los mismos que en los DFDs.

- **Procesos, entidades externas y almacenes de datos.** Serán los mismos y tendrán el mismo significado que en el DFD al que corresponden. Se incluyen en el DFC básicamente como referencia, para mostrar a qué procesos afectan los mecanismos de control que estamos describiendo.
- **Flujos de control.** Se representan mediante trazos discontinuos y modelan el flujo de información de control en el sistema. Habrá procesos o entidades externas que generen información de control y otras que la consuman. La única diferencia es que los flujos de control transportan señales discretas (normalmente lógicas o de tipo enumerado) y son impulsos o eventos, es decir tienen una duración instantánea, mientras que los flujos de datos pueden transportar cualquier tipo de datos (posiblemente estructuras de datos complejas) y pueden transformar datos de manera continua.
- **Almacenes de control.** Se representan igual que los almacenes de datos pero con trazos discontinuos. Permiten almacenar información de control, para ser utilizada posteriormente.
- **Ventanas a especificaciones de control.** Se representan mediante barras. Estas ventanas reciben y emiten flujos de control y representan la transformación de flujos de control en el sistema.

Los procesos de un DFC no representan procesamiento ni transformación de los flujos de control (lo que se hace en las CSPECs) ni tampoco representan los estados del sistema (que se representan en los DEs). Simplemente representan a los mismos procesos de los DFDs, y lo que indica el DFC es simplemente qué flujos de control reciben o generan estos procesos.

Otro detalle importante es que un flujo de control que entra en un proceso no indica que ese proceso se active mediante ese flujo. La activación y desactivación de procesos se indica en la CSPEC. Un flujo de control que entra en un proceso puede indicar dos cosas: bien que va a ser utilizado como una dato más para que el proceso lleve a cabo su función de transformación, o bien que va a ser utilizado para controlar alguno de los procesos en los que se descompone éste. Del mismo modo, un flujo de control que sale de un proceso indica simplemente que ha sido generado por éste e intervendrá en el control del comportamiento de algún otro.

Las especificaciones de control figuran normalmente en un nivel alto de la jerarquía de diagramas. Se encargan de controlar el funcionamiento del sistema, activando o desactivando sus procesos. A bajo nivel, mucho procesamiento de control puede representarse en las primitivas de proceso sin que esto influya en los resultados del análisis.

Cómo separar datos y control.

De entre todos los flujos de información que maneja un sistema software, ¿cómo podemos clasificar unos como datos y otros como control? y ¿qué procesamiento corresponde a procesos de datos y qué procesamiento corresponde al control del

sistema? (los procesos de control no existen como tales en la notación H & P, sino que están incluidos en las CSPECs).

No hay unas normas estrictas para distinguir entre unos y otros, y en algunos casos se puede hacer de forma arbitraria. Como criterio de decisión utilizaremos el siguiente:

Modelaremos como señales de control únicamente aquéllas que intervengan en la activación y desactivación de algunos de los procesos del sistema. El resto las modelaremos como datos.

En determinadas situaciones, un elemento de información determinado se utiliza como dato en un proceso y como control en otro. En este caso, podemos modelarlo como dato (trazo continuo) en el DFD y como control (trazo discontinuo) en el DFC, pero asignaremos a ambos flujos el mismo nombre para mostrar que son el mismo.

Cuándo usar especificaciones de control.

Lo más conveniente es utilizar DFDs siempre que sea posible, puesto que son más sencillos de realizar y de entender (la notación de los DFCs es más oscura y el uso de ambos obliga a mirar los dos a la vez). Sólo para aquellos aspectos del sistema que no podamos modelar con DFDs utilizaremos DFCs. Esto significa reducir el control en la medida de lo posible.

Pese a esto, a la hora de hacer el análisis de requisitos del sistema existe la tendencia de profundizar demasiado en el modelo de control del sistema. Con frecuencia se especifican detalles de implementación como flags, contadores, llamadas a procedimientos e interrupciones, que no tienen nada que ver con los requisitos del sistema. En el análisis de requisitos nos debemos centrarnos en el **modelo abstracto o lógico** del sistema más que en su implementación. Según esto el modelo de control debe ser usado para describir la lógica que controla los procesos principales del sistema y no para describir de forma detallada interacciones entre primitivas de proceso. Esto último puede ser descrito perfectamente utilizando el modelo implícito en los DFDs, que consiste en que cuando un proceso recibe un dato lo procesa de forma inmediata e instantánea. Como este no es el funcionamiento que tendrá el sistema real, existe la tendencia de enviar, junto con el dato, una señal de control que no tiene más objetivo que activar el proceso receptor cuando llegue el dato. Esto no es necesario y no debería hacerse.

Un ejemplo muy común de especificación de control excesiva es el caso en que un proceso genera flujos de salida alternativos, de los cuales sólo se usa uno de cada vez. Podemos entonces generar señales de control para activar/desactivar los procesos receptores pero esto no es un requisito del sistema. Desde el punto de vista de los requisitos podemos dejar todos los procesos funcionando en paralelo; sólo procesarán información cuando reciban entradas. Haciendo esto, podemos expresar los requisitos de forma sencilla y dejar las manos libres a los diseñadores para decidir la implementación más adecuada, siempre y cuando se mantengan las transformaciones de entradas en salidas.

¿Para qué sirven los DFCs?

Considerado aisladamente, solo muestran la información de control que existe en el sistema, junto con los procesos y entidades externas que generan y consumen dicha información de control. Para representar el comportamiento del sistema, hay que utilizarlos en combinación con las especificaciones de control.

5.2.4.- Especificaciones de control

Las especificaciones de control (CSPECs) son similares a las PSPECs. En ambos casos, la función de estas especificaciones es definir los detalles procedimentales de cómo se realiza el procesamiento de los flujos de entrada y salida.

Sin embargo, hay una diferencia importante entre PSPECs y CSPECs. Las PSPECs se utilizan para describir las primitivas de proceso: aquéllos procesos del sistema que son lo suficientemente simples como para no necesitar crear un nuevo DFD. Las CSPECs sirven para modelar el comportamiento de un DFC, describiendo cómo se procesan los flujos de control. Habrá entonces como máximo una CSPEC por cada DFC de la jerarquía. La interfaz entre el DFC y la CSPEC se hace a través de las ventanas de control que aparecen en los DFCs.

Las CSPECs muestran cómo, a partir de las señales de control que entran en la ventana, se determina la activación o desactivación de procesos que figuran en el DFC correspondiente. Es decir, muestran cómo se calculan unos flujos de control de salida (los activadores de los procesos) a partir de otros de entrada (las señales de control que entran en la ventana).

Las CSPECs se caracterizan mediante sistemas combinacionales o, más frecuentemente, mediante sistemas secuenciales. Si son combinacionales su representación se hace mediante Tablas de Decisión (tablas de verdad que indican cómo se calculan las señales de salida en función de las de entrada) o mediante Tablas de Activación de Procesos (iguales que las anteriores, en las que se indica para cada proceso del DFC si está activo o inactivo ante cada combinación de las señales de entrada); si son secuenciales se representan mediante Diagramas de Estados, que no son más que autómatas finitos.

Al hablar de los DFCs habíamos dicho que los flujos de control pueden figurar como salida de alguno de los procesos del sistema. En este caso se denominan condiciones de datos. Como consecuencia del procesamiento de los flujos de datos de entrada del proceso (procesamiento que se describe en la PSPEC asociada al proceso o a alguno de sus descendientes) se genera un flujo de control, que va a intervenir en el comportamiento de otro proceso del sistema. Estos flujos son el único enlace que se establece del modelo de procesos al modelo de control.

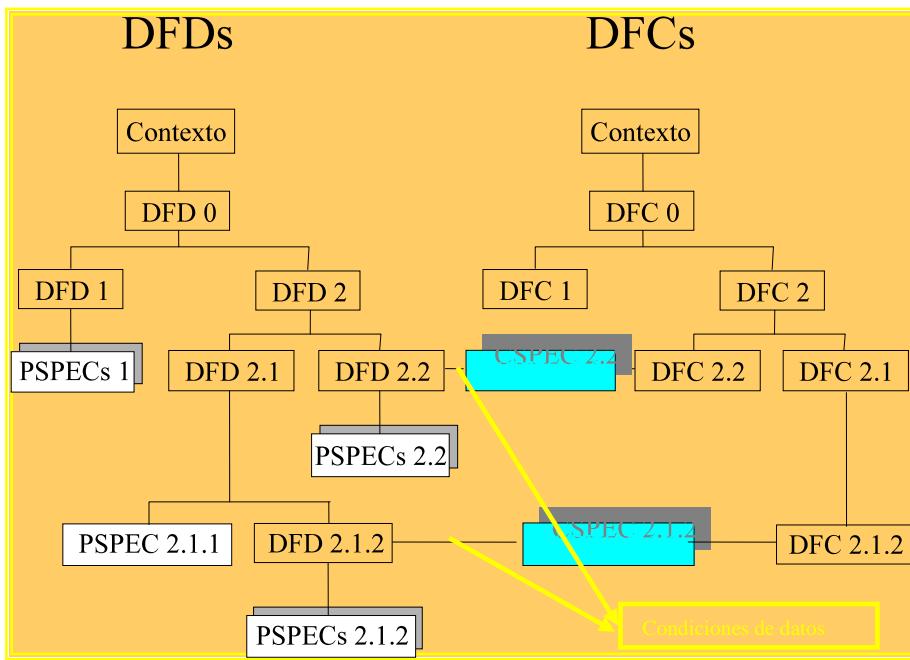


Figura 5.4.- Relación entre DFD y DFC

Condiciones de datos

Ej. El proceso *Comprobar Saldo* procesa *Número de Cuenta* e *Importe* y genera dos flujos de control: *Aceptar Operación* y *Rechazar Operación*.

Resumiendo, las condiciones de datos son flujos de control generados por un proceso del sistema. Figuran en el DFC y no en el DFD, pero es la PSPEC la que define cómo se calcula su valor a partir de los flujos de entrada del proceso.

Ventanas de control.

Estas condiciones de datos, junto con otros flujos de control provenientes del exterior del sistema (de las entidades externas) son los que deciden el comportamiento del sistema. Este comportamiento se define en las CSPECs que indican cómo se generan unos flujos de control a partir de otros o, más concretamente, cómo se activan o desactivan los procesos.

Para hacer referencia a que un flujo de control se calcula a partir de otros y a que esta transformación está definida en la CSPEC se utilizan las ventanas de control en los DFCs. Estas ventanas se representan mediante barras, donde entran y salen flujos de control. Las ventanas de control no están etiquetadas, puesto que todas las que aparecen en un determinado DFC hacen referencia a una única CSPEC que define cómo se realizan estas transformaciones.

Ej. El cajero automático utiliza dos flujos de control (*Saldo suficiente e Importe disponible en cajero*) para calcular los flujos de control *Activar Aceptar Operación* y *Activar Rechazar Operación*.

En un DFC podemos utilizar tantas ventanas de control como sea necesario con objeto de que el diagrama sea claro y todas ellas hacen referencia a la misma CSPEC.

Activadores de procesos.

El objetivo del modelo de control es definir el comportamiento del sistema, es decir, cuándo se activan o desactivan los procesos. Estas activaciones y desactivaciones se modelan mediante unos flujos de control especiales, denominados Activadores de procesos. Estas señales de control toman sólo dos valores: On y Off. Los flujos activadores de procesos tienen el mismo nombre que los procesos que controlan, por lo que no suelen representarse en el DFC, salvo con fines de claridad (identificar qué procesos tienen activador).

En general, los flujos de control que entran en un proceso no sólo sirven para activarlo. Se utilizan también para transformar los flujos de entrada en flujos de salida según se describe en la PSPEC correspondiente o para intervenir en el control de algún subprocesso de éste.

Dado que los modelos del sistema tienen estructura jerárquica, consideraremos que un proceso está activo **sólo si todos sus antecesores (en una jerarquía superior) están activos**. Esto es lógico si tenemos en cuenta que los descendientes de un proceso son subprocessos que están incluidos en él. Cuando se desactiva un proceso, el sistema se comporta como si este proceso y **todos sus descendientes** no existiesen, y se considera que sus salidas toman valor nulo.

Un proceso que no tiene activador se considera activo siempre que sus antecesores lo estén, y responde a los datos de entrada según le van llegando (este es el modelo de control implícito).

Relaciones detalladas entre DFDs, DFCs, PSPECs y CSPECs

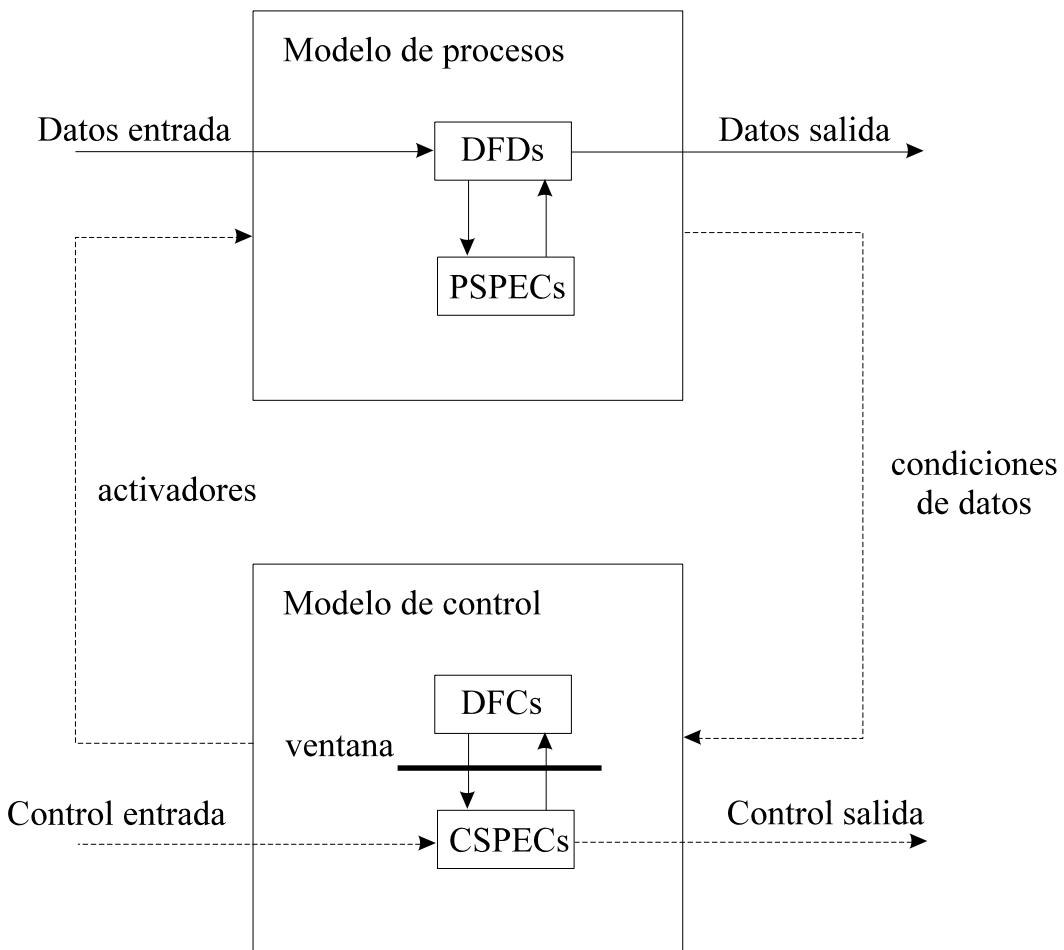


Figura 5.5.- Detalle del funcionamiento de las ventanas a especificaciones de control

En resumen, por cada DFD dibujaremos un DFC gemelo si alguno de los procesos del DFD genera o consume información de control. Además, si alguno de los procesos del DFD es activado o desactivado de forma no implícita, éste DFC llevará una ventana de control y haremos una CSPEC para indicar cuándo se realiza la activación y desactivación.

Una vez que hemos establecido la necesidad de utilizar CSPECs, debemos estudiar si podemos representarlas mediante un sistema combinacional (lo que podremos hacer si la función de control depende únicamente de los valores actuales de las señales de control disponibles) o si necesitamos un sistema secuencial (si la función de control depende de valores anteriores de las señales de control, que ya no están disponibles o si el comportamiento del sistema depende de su evolución anterior. Sólo en este caso utilizaremos CSPECs en forma de DEs.

5.2.5.- Diagramas de estados.

Los sistemas complejos suelen presentar la propiedad de que los eventos pasados influyen en su comportamiento. Estos cambios no se limitan a cambios en los valores de

los flujos de salida, sino que las computaciones pueden cambiar totalmente o incluso desaparecer, y el sistema se comporta de una forma completamente diferente a lo largo del tiempo. Este tipo de comportamiento es difícil de representar utilizando un DFD, pero podemos utilizar técnicas basadas autómatas secuenciales. La forma más habitual de representar autómatas secuenciales es utilizando DEs.

Hay dos modelos muy conocidos de autómatas secuenciales: las máquinas de Moore y las máquinas de Mealy. En las primeras, las salidas se asocian con los estados, mientras que en las segundas se asocian con las transiciones. En los DEs utilizaremos indistintamente uno u otro, incluso un modelo mixto donde sea necesario.

El DE representa el comportamiento de un sistema, mostrando los estados en los que puede estar y los sucesos que hacen que el sistema cambie de estado. Además, el DE indica qué acciones se realizan cuando un sistema cambia de estado y qué actividades se realizan mientras el sistema está en un estado.

Elementos de un DE

Los elementos que aparecen en el DE son dos, Estados y Transiciones:

Estados. Representados mediante rectángulos de esquinas redondeadas. Los estados muestran los distintos modos de comportamiento, escenarios o situaciones en que puede encontrarse el sistema. Cada estado representa un periodo de tiempo en el que el sistema muestra un cierto comportamiento. Generalmente los estados se asocian a la realización de un proceso o a un grupo de procesos del DFD. En otros casos, los estados representan al sistema *esperando* la ocurrencia de un determinado suceso (una petición de servicio o una entrada de datos, p.ej.).

Los estados tienen un nombre que los identifica y pueden ir etiquetados con una **actividad**: aquellos procesos que están activos mientras el sistema esté en dicho estado. Las actividades se corresponden con las señales de salida de una máquina de Moore.

Uno de los estados será el **estado inicial**, aquél en el que se sitúe el sistema cuando comience su funcionamiento (cuando se arranque el programa o se encienda el interruptor general). Un DE puede contener también **estados finales**, de los cuales no se salga mediante ninguna transición. Los estados iniciales y finales están marcados en el DE.

Transiciones. Muestran las evoluciones posibles entre los estados de un sistema y se representan mediante flechas. Indican cuándo *evoluciona* el sistema de un estado a otro. Las transiciones van etiquetadas con dos elementos: la **condición** que hace que se dispare la transición y la **acción** que se produce como consecuencia de la transición. Las acciones corresponden con las señales de salida de una máquina de Mealy.

Las condiciones son generalmente condiciones de datos (esto es, flujos de control) y en este caso se denominan **eventos**, aunque para evitar tener que introducir señales ficticias en el modelo del sistema podemos establecer condiciones asociadas a flujos de datos (la ocurrencia de una determinada entrada de datos, o el que una señal tome determinado valor). En este caso se denominan **guardas** y se representan entre corchetes. Según esto la condición de disparo de una transición está formada por

guardas y/o eventos. Para que una transición se dispare deben cumplirse sus guardas y producirse los eventos con los que esté etiquetada.

Las acciones consisten en generar un suceso o flujo de control o en activar un proceso del DFD (asignar valor *On* a un activador de procesos).

Tanto los eventos como las acciones son flujos de control, y deben figurar en el DFC (exceptuando los activadores de procesos, que normalmente no aparecen en los DFCs).

En el modelo de comportamiento del sistema consideraremos que las transiciones se realizan de forma instantánea, es decir, que al sistema no le lleva tiempo cambiar de estado. Esta no será la situación real del sistema una vez que lo implementemos, pero ahora estamos intentando establecer un modelo lógico o abstracto del sistema.

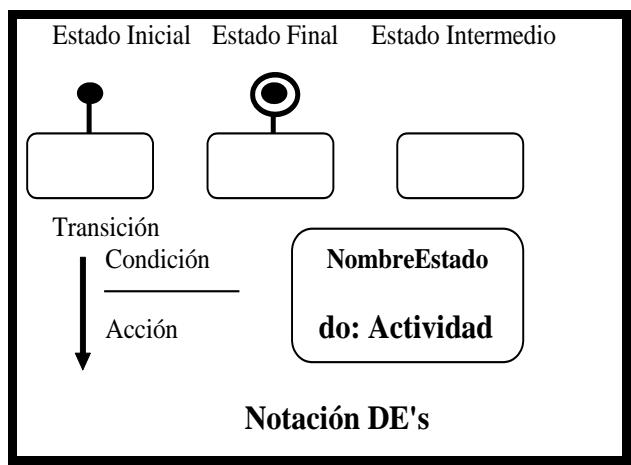


Figura 5.6.- Notación para los diagramas de estado

Como vemos, en el DE podemos representar las computaciones de dos formas distintas: como **acciones** (de duración idealmente instantánea con respecto a la escala de tiempo que utiliza el modelo) o como **actividades** (de duración prolongada en el tiempo, durante todo el intervalo de tiempo en el que el sistema permanezca en el estado correspondiente).

Con respecto a la activación de procesos, si realizamos ésta en un estado, consideraremos que dicho proceso permanece activo mientras el sistema permanece en dicho estado, desactivándose cuando lo abandone. Si, por el contrario, realizamos la activación en la transición supondremos que el proceso sigue activo hasta la siguiente activación, es decir, mientras el sistema permanezca en el estado de salida de la transición. Como se puede apreciar, ambos casos son equivalentes, y no es necesaria la desactivación explícita de procesos.

El DE muestra, por tanto, el comportamiento del sistema. Estudiando el DE podemos comprobar si hay lagunas en el comportamiento especificado: si hay estados de los que no se sale mediante ninguna transición o hemos considerado todas las transiciones posibles (esto es si en un estado hemos previsto todos los sucesos que pueden ocurrir y causar una evolución del sistema).

Esto último es muy importante: supongamos un sistema con tres estados. Del estado inicial A, pasamos al estado B pulsando la tecla F1, y del B pasamos al C pulsando F2. El DE de la figura modela este comportamiento, pero no podemos quedarnos en modelar el comportamiento del sistema basándonos en un *comportamiento correcto del usuario*. ¿Qué sucede si el usuario pulsa dos veces la misma tecla? ¿O si pulsa cualquier otra? Si no especificamos el comportamiento del sistema ante estos eventos, posiblemente los programadores no crearán código para estas situaciones y el sistema final podrá presentar un comportamiento incontrolado, ante un usuario poco experto o malicioso. No podemos hacer suposiciones sobre el buen uso del sistema y tendremos que intentar dejarlo siempre todo bien atado.

Nosotros utilizaremos los DEs para modelar el comportamiento de todo el sistema o una parte de él, incluyendo estos diagramas en las CSPECs. Por tanto, los DEs van a ir asociados a un par DFD/DFC, indicando cómo se procesan los flujos de control que entran en las ventanas de control de los DFCs y cómo se activan los procesos que figuran en los DFDs. Los flujos de control que entran en la ventana van a ser condiciones de las transiciones del DE. Los flujos que salen de la ventana serán establecidos en las acciones y actividades de dicho DE.

Composición de DEs.

En algunos sistemas, los procesos de un DFD se activan y desactivan de forma más o menos independiente. Este es el caso de un sistema con dos o más procesos concurrentes que no interactúan entre sí o lo hacen de forma limitada. En estos casos realizaremos un DE compuesto, es decir formado por un DE para cada proceso o grupo de procesos que se comportan de forma independiente.

Ej. En la máquina expendedora (ejercicio 1), el control de la aceptación de monedas y el control de la selección de productos son prácticamente independientes. La única interacción entre ellos se produce a través del almacén IMPORTE. El DE se compone entonces de dos: uno para cada subsistema.

Anidamiento de DEs.

Igual que los DFDs y DFCs, los DEs también pueden anidarse, en caso de que sea necesario. Esto será útil cuando el DE sea complejo, y sea conveniente mostrarlo en varios niveles de abstracción o cuando un grupo de estados tienen un comportamiento común (reaccionan de una forma determinada ante un mismo evento (p. ej. un evento de error o de cancelación). En estos casos utilizaremos DEs anidados.

En cualquier caso, en DE anidado va a contener estados que se descomponen en un DE completo. Cualquier transición de entrada en dicho estado nos lleva al estado inicial del DE anidado. Los estados finales del DE anidado nos llevan a la transición de salida correspondiente del estado anidado.

Acciones dentro de los estados.

La inclusión de acciones de duración instantánea dentro de los estados amplía el modelo de automátas, dándole mayor expresividad y evitando duplicaciones innecesarias:

- ◆ **Acciones de entrada.** Si una determinada acción ha de realizarse en todas las transiciones de entrada de un estado determinado, podemos ponerla como acción de entrada de dicho estado, evitando repetirla en cada transición.
 - ◆ **Acciones de salida.** Igualmente, si una determinada acción ha de realizarse en todas las transiciones de salida de un estado, la pondremos como acción de salida del mismo, evitando repetirla en cada transición.
 - ◆ **Eventos internos.** Si, mientras el sistema está en un estado determinado, puede producirse un evento que lleve asociada una determinada acción, pero que no provoque un cambio de estado, podemos representar esto como un evento interno del estado. En este caso no se realizarán las acciones de entrada y de salida del estado, puesto que se considera que el sistema no cambia de estado al atender dicho evento.

5.2.6.- Redes de Petri

Las redes de Petri fueron creadas por Carl Adam Petri en 1963. Es una técnica muy apropiada para la descripción del control en sistemas de comportamiento asíncrono y Concurrente.

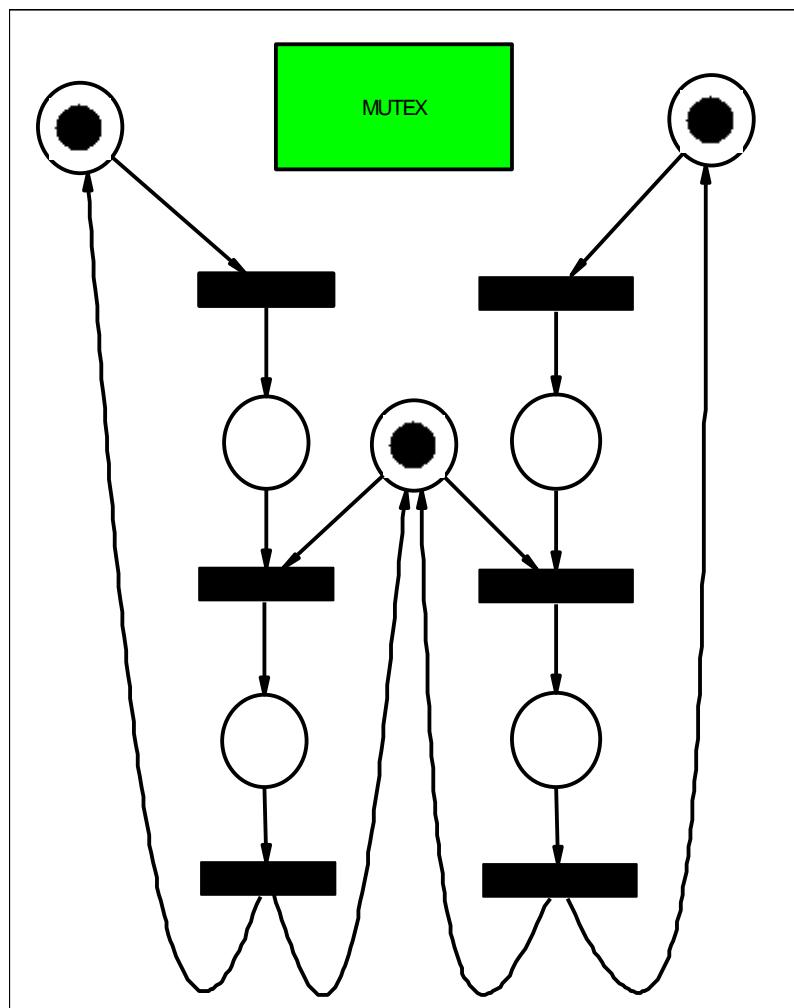


Figura 5.7. Representación gráfica de una red de Petri

Una red de Petri es un modelo compuesto por los siguientes elementos:

- Un conjunto finito de lugares representados por círculos.
- Un conjunto finito de transiciones, representados por segmentos.
- Un conjunto finito de conexiones o arcos de un lugar con una transición o viceversa, representadas por flechas.

Las redes de Petri adquieren un estado en función de su marcado, esto es, de la asignación de un número no negativo de marcas (también llamadas tokens) a cada lugar de la red. Las marcas se representan gráficamente mediante pequeños círculos negros (véase figura 5.7). Se puede considerar que cada marcado diferente representa un estado distinto. Evidentemente, toda red debe partir de un marcado o estado inicial a partir del cual evoluciona. La evolución del estado de la red se produce a través de la llamada regla de disparo de transiciones que consiste en lo siguiente:

- Cada transición consta de lugares de entrada (aquellos desde los cuales llega una flecha a la transición) y lugares de salida (a los que llega una flecha desde la transición). Una transición está habilitada cuando existe, al menos, una marca en cada uno de sus lugares de entrada.
- Una transición habilitada puede dispararse. Si se dispara, se consume (se elimina) una marca de cada lugar de entrada y se produce (se añade) una marca en cada lugar de salida.

Las redes de Petri son modelos muy genéricos que admiten muchísimas variantes. Las principales son las siguientes:

- Redes con valores en los arcos, es decir, que en cada disparo provocan el consumo de más de una marca.
- Redes que identifican las marcas, bien creando distintos tipos a los que se asignan colores (redes coloreadas) o bien asignando valores concretos a las marcas e incluyendo en condiciones relativas a dichos valores para el disparo de las transiciones (redes de predicados y funciones),
- Redes en las que se asignan tiempos a las transiciones o a los lugares. Estas redes se denominan genéricamente redes temporizadas y, a su vez, admiten muchas otras variantes en función de cómo se realice la asignación de tiempos.

De hecho, los diagramas de transición de estados podrían considerarse como un determinado tipo de redes de Petri. No obstante, en ambos modelos es importante resaltar la necesidad de una etapa de verificación de los mismos. Los modelos deberían tener las dos propiedades siguientes:

- **Limitación**, es decir, cualquiera que sea la evolución desde cualquier marcado inicial no sobrepasa un determinado límite finito de marcas en cada lugar. Esta propiedad es muy importante, porque el almacenamiento de las marcas implica un consumo de memoria, que no es un recurso infinito.
- **Vivacidad**, es decir, cualquiera que sea la evolución desde cualquier marcado inicial la red no se bloquea, siempre puede dispararse alguna transición.

La comprobación de estas propiedades para los posibles marcados iniciales suele ser bastante compleja.

5.2.7.- Diagramas Entidad/Relación.

El tercer punto de vista que podemos considerar para hacer un modelo del sistema software, es el **punto de vista de los datos**. La notación básica de los DFDs, que representan los datos mediante almacenes de datos, es suficiente cuando la información que fluyen entre los procesos es relativamente simple. Sin embargo, las aplicaciones de software de gestión y cada vez más las aplicaciones de ingeniería y de tiempo real, utilizan colecciones de datos complejas. En estos casos no basta con saber en detalle qué información contiene cada almacén de datos, sino qué relaciones existen entre estos almacenes. Para ello es necesario desarrollar un modelo de datos del sistema, lo que haremos mediante Diagramas Entidad/Relación (DER). Para seguir la metodología nos remitimos a los apuntes de la asignatura de Bases de Datos.

5.2.8.- Diccionario de datos².

El diccionario de datos o diccionario de requisitos es el lugar donde van a estar contenidas las definiciones de todos los elementos que aparecen en los distintos diagramas del sistema, y sirve por tanto para establecer la relación entre los distintos modelos del mismo (procesos, comportamiento y datos). El diccionario de datos permite establecer que el analista y el usuario entienden de la misma forma cada uno de los elementos de los diagramas.

El análisis del sistema software no va a estar completo porque hagamos una serie de diagramas DFD, DFC, DER y DE. En cada representación los objetos de datos y/o elementos de control juegan un papel importante. Por consiguiente, es necesario proporcionar un enfoque *organizado* para representar las características de cada objeto de datos y elemento de control. Esto se realiza con el diccionario de datos.

Se ha propuesto el *diccionario de datos* como gramática casi formal para describir el contenido de los objetos definidos durante el análisis estructurado. Esta importante notación de modelado ha sido definida de la siguiente forma:

Es un listado organizado de todos los elementos de datos que son pertinentes para el sistema, con definiciones precisas y rigurosas que permiten que el usuario y el analista del sistema tengan una misma compresión de las entradas, salidas, de las componentes de los almacenes y de los cálculos intermedios

Actualmente, casi siempre se implementa el diccionario de datos como parte de una «herramienta CASE de análisis y diseño estructurados». Aunque el formato del diccionario varía entre las distintas herramientas, la mayoría contiene la siguiente información:

² Pressman5, pag. 215

- **Nombre:** Nombre principal descriptivo único del elemento de datos o de control, del almacén de datos, o de una entidad externa
- **Alias:** Otros nombres
- **Dónde se usa/Cómo se usa:** Un listado con
 - Origen, Destino del flujo de datos.
 - Procesos que usan el dato y como lo usan
 - Flujo de datos de entrada o salida.
 - Almacén de datos.
 - Entidad externa.
- **Descripción del contenido:** El contenido representado según alguna notación.
- **Comentarios adicionales** Tipo de datos (Archivo, Pantalla, Reporte, Interno entre procesos), valores implícitos, si es un flujo volumen por unidad de tiempo...

Una vez que se introducen en el diccionario de datos un nombre y sus alias, se debe revisar la consistencia de las denominaciones. Es decir, si un equipo de análisis decide denominar un elemento de datos recién derivado como *xyz*, pero en el diccionario ya existe otro llamado *xyz*, la herramienta CASE que soporta el diccionario muestra un mensaje de alerta sobre la duplicidad de nombres. Esto mejora la consistencia del modelo de análisis y ayuda a reducir errores.

La información de «dónde se usa/cómo se usa» se registra automáticamente a partir de los modelos de flujo. Cuando se crea una entrada del diccionario, la herramienta CASE inspecciona los DFD y los DFC para determinar los procesos que usan el dato o la Información de control y cómo lo usan. Aunque esto pueda no parecer importante, realmente es una de las mayores ventajas del diccionario. Durante el análisis, hay una corriente casi continua de cambios. Para proyectos grandes, a menudo es bastante difícil determinar el impacto de un cambio. Algunas de las preguntas que se plantea el ingeniero del software son «¿dónde se usa este elemento de datos? ¿qué mas hay que cambiar si lo modificamos? ¿cuál será el impacto general del cambio?». Al poder tratar el diccionario de datos como una base de datos, el analista puede hacer preguntas basadas en «dónde se usa/cómo se usa» y obtener respuestas a peticiones similares a las anteriores. Para desarrollar una descripción de contenido se sigue la siguiente notación:

- = **Composición:** “está compuesto de”
- + **Inclusión:** “y”
- [a | b | c] **Selección:** situación disyuntiva
- () **Opción:** elemento opcional
- { }n **Iteración:** repetición de un elemento n veces
- *texto* **Comentario:** aclarativo de una entrada del DD
- @ **Identificador:** Marca los campos que identifican ocurrencia

La notación permite al ingeniero del software representar una composición de datos en una de las tres alternativas fundamentales que pueden ser construidas

1.- Como una secuencia de elementos de datos.

- 2.- Como una *selección* de entre un conjunto de elementos de datos.
- 3.- Como una *agrupación* repetitiva de elementos de datos, Cada entrada de elemento de datos que aparezca como parte de una secuencia, una selección o una repetición puede a su vez ser otro elemento de datos compuestos que necesite un mayor refinamiento en el diccionario.

Para ilustrar el uso del diccionario de datos especificamos el elemento de datos número de teléfono. ¿Qué es exactamente un número de teléfono? Puede ser un número local de siete dígitos, una extensión de 4 dígitos, o una secuencia de 25 dígitos para llamadas de larga distancia. El diccionario de datos nos proporciona una definición precisa de número de teléfono en cuestión. Además, indica dónde y cómo se usa este elemento de datos y cualquier información adicional que le sea relevante. La entrada del diccionario (le datos comienza de la siguiente forma:

Nombre: Número de teléfono
 Alias:
 Dónde se usa / Proceso en el que es entrada
 Cómo se usa: Proceso en el que es salida
 Descripción: prefijo + número de acceso
 Prefijo = *secuencia de 4 dígitos comenzando en 0*
 Número de acceso = *secuencia numérica de cualquier tamaño*

Para grandes sistemas basados en computadora el diccionario de datos crece rápidamente en tamaño y en complejidad. De hecho, es extremadamente difícil mantener manualmente el diccionario. Por esta razón, se deben usar herramientas CASE.

5.2.9.- Comprobaciones a realizar sobre una especificación estructurada³

Una vez esté realizada la especificación estructurada, formada por el conjunto de DFD, el diccionario de datos y las especificaciones de proceso, es necesario revisarla. Como indica Yourdon [YOURDON, 1985], es conveniente hacer la revisión en base a cuatro aspectos; compleción, integridad, exactitud y calidad, por los que se comprueba:

- **Compleción:** Si los modelos de la especificación estructurada son completos.
- **Integridad:** Si no existen contradicciones ni inconsistencias entre los distintos modelos.
- **Exactitud:** Si los modelos cumplen los requisitos del usuario.
- **Calidad:** El estilo, la legibilidad y la facilidad de mantenimiento de los modelos producidos.

Las metodologías de desarrollo de software proponen la realización de revisiones en puntos concretos del desarrollo. Cuando el tipo de revisión es formal (por

³ Piattini 2003

ejemplo, una inspección) es necesario establecer una lista de comprobación para guiar a los revisores en la reunión de revisión. Cuando se utilice una herramienta CASE, muchas de estas cuestiones se pueden resolver de forma automática. En la Tabla, se muestra un ejemplo de lista de comprobación en la que se formulan preguntas en base a los aspectos anteriormente citados y se señalan en negrita aquellas que no pueden ser resueltas por las herramientas.

Una lista de comprobación (en inglés "checklist") es una serie de preguntas sencillas con dos tipos de respuesta posibles. sí o no. Es un elemento clave para la detección de defectos en las inspecciones de software por la cual se guían los revisores, que revisan el producto centrándose en las preguntas de dicha lista.

	PREGUNTA	Sí	No
C	Todos los componentes tienen nombres		
C	Todos los procesos tienen números		
C	Todos los procesos primitivos tienen una especificación de proceso asociado		
C	Todos los flujos están definidos en el DD		
C	Todos los elementos de datos están definidos		
1	Hay elementos definidos en el DFD no incluidos en el DD		
1	Los almacenes de datos representados en los DFD están definidos en el DD		
1	Los elementos de datos referenciados en las especificaciones de proceso están definidos en el DD		
1	Los flujos de datos de entrada y salida de un proceso primitivo se corresponden con las entradas y salidas de la especificación de proceso		
1	Hay errores de balanceo		
1	Hay procesos que tienen sólo entradas o sólo salidas		
1	Por cada proceso se cumple la regla de conservación de datos		
1	Hay flujos de entrada superfluos a un proceso		
1	Hay flujos de controlo flujos de datos como activadores de procesos		
1	Los procesos pueden generar los flujos de salida a partir de los de entrada más una información local al proceso		
1	Hay pérdida de información en los procesos		
1	Hay almacenes sólo con entradas o sólo con salidas		
1	Hay conexiones incorrectas entre los elementos del DFD		
1	Hay almacenes locales		
1	Es correcta la dirección de las flechas de los DFD		
1	Existen redes desconectadas		
E	Cada requisito funcional del usuario tiene asociado uno o más procesos primitivos en los DFD		
CA	El diagrama es claro (posición correcta de las etiquetas, existencia de cruces de línea, etc.)		
CA	Hay nombres de componentes con poca significación		
CA	Hay muchos flujos de entrada y salida (complejidad de interfaz alta) en procesos primitivos		

5.3.- Consistencia entre modelos.

En los apartados anteriores hemos visto cómo desarrollar modelos de procesos, de comportamiento y de datos de un sistema, cada uno de estos modelos se centra en un determinado aspecto del sistema. También hemos visto cómo definir todos los elementos de estos modelos en un diccionario de datos y una serie de reglas para determinar la consistencia de los diferentes diagramas de se utilizan en cada modelo.

El conjunto de modelos tiene como misión dar una visión global del sistema, desde diversos puntos de vista. Los modelos pueden parecer muy distintos pero todos

ellos representan el mismo sistema por lo que deben ser consistentes. En este apartado vamos a ver como podemos asegurar la consistencia entre los distintos modelos. Los principales modelos sobre los que efectuaremos comprobaciones son los siguientes:

- **Dimensión de función:** en donde tenemos la especificación estructurada compuesta de los diagramas de flujo de datos, diccionario de datos y especificaciones de procesos.
- **Dimensión de información:** diagrama entidad/interrelación (DE/R), o diagrama de estructura de datos (DED).
- **Dimensión del tiempo:** lista de eventos, diagramas de flujo de control y especificaciones de control.

Conforme hemos descrito estos modelos hemos ido señalando algunas normas de consistencia que ahora recogeremos juntas:

Consistencia entre el modelo de procesos y el diccionario de datos.

Las reglas son las siguientes:

- **Cada elemento de los DFDs debe estar definido en el DD.** Los almacenes y flujos de datos tienen que estar definidos en el diccionario.
- **Cada dato elemental del DD tiene que estar incluido en algún flujo de datos.** Si no fuese así el diccionario contendrá datos elementales que no se usan en el sistema. Además, si el sistema debe memorizar alguno de estos datos, debe figurar también en un almacén de datos de los DFDs.

Una excepción a esta regla serían las variables locales de las primitivas de proceso, que sólo aparecen en las PSPECs, pero definir estas variables locales en el DD no es lo habitual.

Consistencia entre el modelo de control y el diccionario de datos.

Las reglas son exactamente las mismas: cada flujo y almacén de control de los DFC y cada señal utilizada en las condiciones o acciones del DE, debe estar definido en el diccionario. Por otro lado, cada señal de control definida en el diccionario tiene que estar incluida en un flujo de control de los DFCs o tiene que ser una variable local de la CSPEC.

Consistencia entre el modelo de datos y el diccionario de datos.

- Los objetos y almacenes de datos del DER tiene que estar definidos en el DD. Para cada uno de ellos hay que describir los datos elementales que contienen.

Consistencia entre el modelo de procesos y el modelo de datos.

El DER muestra el sistema desde un punto de vista muy distinto al de los DFDs, sin embargo hay una serie de reglas que deben cumplirse para que ambos sean consistentes:

- **Cada almacén de los DFD debe corresponderse con un objeto o relación del DER.** Ambos elementos representan necesidades de almacenamiento de información del sistema.
- **Los nombres de los elementos del DER y de los almacenes de los DFDs tienen que corresponderse.** El convenio que seguiremos es usar nombres en plural para los almacenes de datos (p.ej. Clientes) y en singular para los objetos (p.ej. Cliente).
- Las entradas del diccionario deben aplicarse tanto a los almacenes de datos como a los objetos y relaciones correspondientes.

Consistencia entre el modelo de procesos y el modelo de control.

- **Cada DFC y cada CSPEC están asociados a un DFD.** (El recíproco no es necesariamente cierto). Los procesos, entidades externas y almacenes del DFD deben aparecer en el DFC correspondiente.
- **Cada estado del DTE se asocia a un proceso o grupo de procesos del DFD, activos durante ese estado, o al sistema esperando que se produzca algún evento.** De acuerdo a esto, los nombres de estados y procesos deben estar relacionados: para los procesos utilizaremos infinitivos (p.ej. Comprobar Saldo Cliente) y para los estados, gerundios (p.ej. Comprobando Saldo Cliente).

A estas reglas cabe añadir:

Plano información-función

- Comprobar que todos los elementos (o datos elementales) definidos en los diagramas entidad/interrelación están definidos como entradas en el DD, es decir, están en algún flujo de datos o almacén. No tiene por qué ocurrir lo contrario (todos los elementos del DFD en el diagrama E/R), ya que en el diagrama E/R no se representan datos secundarios.
- Realizar la misma comprobación con los diagramas de estructuras de datos. En este caso, éstos pueden contener datos secundarios a efectos de rendimiento del modelo.
- Comprobar que cada entidad e interrelación del DE/R es consultada y actualizada al menos una vez por alguna función primitiva del DFD (lo que implica que habrá que comprobarlo en las especificaciones de proceso).

Plano información-tiempo

- Comprobar que cada entidad del DED tiene asociado un diagrama de historia de la vida de la entidad (HVE). Si se hace con el DE/R además de las entidades habrá que comprobarlo con las interrelaciones M:N.
- Comprobar que por cada entidad existe un evento que la crea. Hay que preguntarse el porqué en caso de no haber eventos que la actualicen o eliminen. Para esto, se construye la matriz evento-entidad.

- Comprobar que en las HVE de las entidades «maestro» 26 se tratan las posibles repercusiones que tiene el borrado de dicha entidad sobre las entidades «detalle».

Plano tiempo-función

- Comprobar que existe un proceso primitivo dentro de los DFD que trate cada uno de los eventos identificados en la HVE.

5.3.1.- Técnicas matriciales

Las técnicas matriciales se utilizan principalmente para ayudar a verificar la consistencia entre los componentes de distintos modelos de un sistema, ya sean centrados en las funciones, en la información, o en el aspecto temporal.

	FUNCTION	INFORMACIÓN	TIEMPO
FUNCTION			
INFORMACIÓN	Matriz Entidad/Función	Matriz Entidad/Entidad	
TIEMPO		Matriz Evento/Entidad	

Hay que resaltar que todas las técnicas matriciales no tienen que encuadrarse en esta tabla. Así, la matriz de papeles de usuario/función utilizada por la metodología SSADM relaciona los usuarios que van a utilizar el sistema con determinadas funciones del mismo. Cada celda de la matriz indica un diálogo, es decir, un conjunto de pantallas mediante las cuales el usuario se comunica con una función del sistema. En SSADM es necesario realizar este estudio antes de comenzar a diseñar la interfaz del usuario mediante la técnica de diseño del diálogo.

Matriz entidad/función

Esta matriz visualiza las relaciones existentes entre las funciones que lleva cabo un sistema y la información necesaria para soportar las mismas.

Los elementos de las filas pueden ser *entidades*, *interrelaciones*, *entidades asociativas* y *subtipos*, presentes en un diagrama entidad/interrelación. Los elementos de las columnas están formados por las funciones del sistema, que pueden ser funciones de alto nivel representadas en un DFD o bien las funciones primitivas del conjunto de DFD.

En cada celda se incluyen las posibles acciones que puede realizar una función sobre las ocurrencias de las entidades, interrelaciones, etc. Estas acciones pueden ser crear o insertar (I), leer (L), modificar o actualizar (M) y borrar (B).

Funciones Entidades	Gestionar presupuesto cliente	Gestionar Cliente	...
CLIENTE	L	I, M, B	
PRESUPUESTO	I, M, B		
...			

Matriz entidad/entidad

Es una matriz que ayuda a ver las relaciones que tiene una entidad con otras del modelo entidad/interrelación. Es especialmente útil cuando el número de entidades es grande. En el caso de que exista una interrelación entre dos entidades se incluye en la matriz una X o el nombre de la interrelación.

Entidad Entidad	CLIENTE	PRESUPUESTO	...
CLIENTE		Tiene	
PRESUPUESTO			
...			

Matriz evento/entidad

En las filas se incluyen todos los eventos, entendiendo como evento un suceso que ocurre en la vida del sistema que supone una alteración de los datos almacenados. En cada columna se incluyen las entidades del diagrama de estructura de datos (véase apartado 75.3). En cada celda se define la alteración causada por el evento, que puede ser la inserción o creación (I), modificación o actualización (M) y borrado (B) de una ocurrencia de la entidad. Hay que comprobar que cada entidad tiene algún evento que la crea, modifica o borra. Si no es así (lo que no necesariamente indica un error), es conveniente preguntarse el porqué. Por el contrario, es necesario que exista por cada entidad un evento que la crea. Además, es necesario asegurarse de que cada evento afecta a la vida de al menos una entidad. Si alguna fila o columna no tiene marca, se habrá cometido un error grave.

Entidades Eventos	CLIENTE	PRESUPUESTO	...
Datos del Cliente	I, M, B		
Datos del presupuesto	I	I, M, B	
...			

5.4.- Metodología del análisis estructurado.

En el apartado anterior hemos visto diversas notaciones para el análisis estructurado de sistemas. Estas notaciones nos permiten modelar el software desde diversos puntos de vista. Vamos a ver ahora como podemos coordinar estos modelos y cuál es el método de trabajo apropiado.

5.4.1.- Fases.

Creación del modelo de procesos.

El punto de partida será el modelo de procesos del sistema. Mediante el uso de DFDs y PSPECs podemos modelar el ámbito de información y ámbito funcional del sistema.

La tarea de análisis consiste básicamente en eso: en analizar o pensar, no en ponerse a dibujar diagramas a lo loco. Por esto, lo primero que tenemos que hacer es estudiar toda la documentación inicial que tengamos del sistema, bien sea una especificación preliminar, o el resultado de las reuniones o entrevistas preliminares con el usuario.

Podemos hacer un análisis gramatical de la información de entrada. Identificando los nombres y los verbos de la especificación preliminar podremos identificar muchos de los componentes del sistema. Podemos clasificar estos elementos en entidades externas (nombres), flujos y almacenes de datos (nombres) y procesos o funciones (verbos). La especificación relaciona estos nombres y verbos entre sí. Esto nos permitirá establecer la relación entre procesos y el resto de los componentes. Podemos hacer listas separadas de cada uno de estos componentes. No serán correctas ni completas, pero es un buen punto de partida.

A continuación, comenzaremos por hacer el DFD de contexto, mostrando el sistema software en relación con las entidades externas con las que interactúa. En este nivel representamos el sistema mediante un sólo símbolo de proceso, no se suelen mostrar almacenes de datos, y se identifican las entradas y salidas principales del sistema (no se detallan todos los flujos de datos, sino que se los agrupa).

A continuación, vamos refinando el DFD de contexto en mayores niveles de detalle. Este es un proceso de descomposición funcional, en el que tenemos que ir descomponiendo el sistema en las distintas funciones que realiza. Hay que aplicar los principios de **acoplamiento mínimo** (máxima independencia) entre procesos distintos y **máxima cohesión** (fuerte conexión funcional entre todo lo que está dentro de un proceso) en cada proceso. Si todos los flujos de datos que aparecen en un DFD son utilizados en todos los procesos del mismo, posiblemente algo irá mal. Según esto, un proceso debe realizar una función clara y coherente.

El número de procesos que aparecen en un DFD debe estar (idealmente) entre 5 y 10. Poner menos llevará a tener que hacer demasiados DFDs y a que alguna de las burbujas no sea más que un saco de procesos (es decir, habrá poca cohesión). Poner más dificultará el conseguir una visión general del diagrama.

Según vamos descomponiendo, tenemos que tener cuidado en mantener la consistencia: consistencia de nombres, de numeración y de flujos de datos, aplicando la regla de balanceo.

No hay que caer en detalles de implementación, el DFD pretende dar una visión de cómo se mueve idealmente la información entre los diferentes procesos, y no en ser un lenguaje de programación gráfico.

Según vamos incluyendo elementos en los DFDs hay que ir definiéndolos en el diccionario de datos del proyecto.

Seguiremos el proceso de descomposición hasta encontrar las primitivas de proceso: estas serán procesos sencillos, con máxima cohesión (realizan una única función). Los flujos de datos que entran y salen de las primitivas constituyen la interfaz de estos procesos. Describiremos estas primitivas mediante PSPECs, a ser posible en pseudocódigo, manteniendo la consistencia de flujos de entrada y salida con respecto a los DFDs.

Las PSPECs no contienen definiciones de datos, por tanto, no podemos ocultar en ellas almacenes de datos. Estos tienen que aparecer en los DFDs.

Creación del modelo de control.

No todos los sistemas necesitan de un modelo de control. En muchos casos, el comportamiento implícito en los DFDs servirá para mostrar cómo se comporta el sistema. Debido a esto, lo primero es determinar si es necesario o no desarrollar un modelo de control.

Aquí hay que tener especial cuidado en no caer en detalles de implementación, y empezar a pensar en qué programas llaman a qué programas o como se sincronizan los procesos. Esta es una tarea más propia del diseño.

Si hemos decidido desarrollar un modelo de comportamiento del sistema, debemos empezar por establecer una jerarquía de DFCs simétrica a la de DFDs. Cada par DFD/DFC contiene los mismos procesos, almacenes de datos y entidades externas, y en la misma posición, para facilitar la identificación.

A continuación, eliminaremos del DFD todos los flujos que transporten información de control: aquella que sirve para determinar el comportamiento del sistema, activando o desactivando procesos. Estos flujos de control los representaremos en los DFCs.

La jerarquía de DFCs será normalmente más corta que la de DFDs. Continuaremos desarrollando DFCs mientras los procesos que aparecen en ellos generen y sean controlados por los flujos de control que vayamos especificando. En aquellos casos en los que el comportamiento del sistema quede reflejado implícitamente en los DFDs no haremos DFCs.

A cada par DFD/DFC le corresponde una CSPEC. Las condiciones de datos generadas por los procesos y las señales de control provenientes del exterior del DFC entrarán en las ventanas de control. De estas ventanas saldrán los activadores de procesos (se pueden representar o no, opcionalmente) y las señales de control que salgan del DFC.

Hay que mantener la consistencia de flujos de control entre las ventanas de control y las CSPECs.

La CSPEC puede ser combinacional, secuencial o compuesta. Elegiremos siempre el modelo más sencillo posible. Las CSPECs combinacionales pueden representarse mediante tablas de decisión y tablas de activación de procesos. Las secuenciales mediante DEs.

Respecto a los DEs, hay que tener cuidado con las posibles omisiones de transiciones. ¿Existe alguna otra forma de llegar a un estado o salir de él?, ¿Están previstas todas las contingencias que pueden presentarse en cada estado?

Creación del modelo de datos.

Si los datos que maneja el sistema tienen una estructura compleja, no bastará con definir en el diccionario el contenido de cada uno de los almacenes de datos. Desarrollaremos un modelo de datos, utilizando DERs, donde se muestren las relaciones que existen entre los datos que maneja el sistema. Como punto de partida de este modelo utilizaremos los almacenes de datos y entidades externas que aparecen en el modelo de procesos.

Consistencia entre modelos.

Una vez se han realizado los distintos modelos del sistema se deberían llevar a cabo las técnicas de consistencia entre modelos discutidas en el apartado anterior.

5.5.- Modelos del sistema: esencial y de implementación.

El modelo esencial del sistema (algunas veces llamado modelo lógico del sistema) es un modelo de lo que el sistema debe hacer con objeto de satisfacer los requisitos del usuario. En el modelo esencial no figura (al menos idealmente) nada acerca de cómo se va a implementar el sistema. Es por tanto, un modelo abstracto del sistema, que supone que disponemos de una tecnología perfecta sin coste alguno.

Este modelo esencial es el resultado de la fase de análisis de requisitos del sistema, y de este tipo son todos los modelos que hemos estado viendo hasta ahora.

El modelo esencial tiene que estar completamente libre de detalles de implementación. Alguno de los errores típicos en los que se cae al hacerlo son:

- **Secuenciar de forma arbitraria los procesos del DFD.** Los procesos de los DFDs deben ser lo más concurrentes posible, no hay que pensar en que el sistema realiza una tarea, y luego otra, y luego otra. El único secuenciamiento de funciones que debe aparecer en los DFDs es el impuesto por la dependencia de datos. Si un proceso P2 recibe como entrada un flujo de datos generado por un proceso P1, no podrá realizarse hasta que P2 acabe. Cualquier otra secuencia es puramente arbitraria, y dependerá más de necesidades de implementación (p.ej. capacidad del ordenador) que de requisitos del sistema.
- **Utilizar ficheros temporales o de backup.** Los ficheros temporales se usan para conectar procesos que no pueden ejecutarse simultáneamente por problemas de capacidad o multiproceso del ordenador, no porque los procesos deban ejecutarse de forma secuencial. Los ficheros de backup se utilizan para evitar perder información en caso de que falle el sistema (sea el fallo hardware, software o humano). Supuesta una tecnología perfecta,

no necesitamos de ninguno de ellos. No son requisitos esenciales del sistema sino de implementación.

- **Utilizar información redundante o derivada.** El incorporar en las bases de datos o en los procesos información redundante tiene más que ver con la eficiencia de la implementación que con los requisitos del modelo. (P.ej. los totales de factura se pueden calcular a partir de las líneas, sin embargo, es normal almacenarlos para evitar tener que calcularlos cada vez que se accede a la factura). En el modelo esencial no usaremos nunca información redundante o derivada.

A partir del modelo esencial, los diseñadores podrían decidir cómo implementar el sistema, usando la tecnología disponible. Podrían decidir cuál va a ser el hardware, la base de datos, el lenguaje de programación, etc. y cómo implementar con estas herramientas los elementos del modelo esencial.

Sin embargo, lo normal es que el cliente proporcione más información que los simples requisitos del sistema, y que decida también sobre detalles de implementación: qué funciones van a ser manuales y cuáles automáticas, el formato de los informes y de las pantallas, requisitos operacionales de velocidad de cálculo o capacidad, etc. Toda esta información que el cliente proporciona al analista, que no se refiere a los requisitos esenciales del sistema, sino a los requisitos de implementación, formarán parte del modelo de implementación del sistema.

El desarrollo de un modelo de implementación es una tarea que está a caballo entre el análisis y el diseño. No puede ser desarrollado sólo por el analista y el cliente pues se necesita el consejo de diseñadores e implementadores sobre las elecciones de hardware y software y sobre la posibilidad de cumplir las restricciones de tiempos de respuesta y capacidades del sistema usando la tecnología elegida. Por otra parte, no puede ser realizado únicamente por diseñadores e implementadores porque el usuario debe definir una gran cantidad de requisitos de implementación, que irán surgiendo en la fase de análisis, y es al analista al que se los describe. (El analista es el principal vínculo entre el cliente y el equipo de desarrollo. El cliente tiene muy poco trato con diseñadores e implementadores).

Este modelo de implementación será una versión revisada y anotada del modelo esencial, donde se especifiquen todos estos detalles adicionales proporcionados por el usuario. Entre estas anotaciones podemos citar:

- **La elección de dispositivos de entrada y salida.** Desde el punto de vista del modelo esencial, el sistema recibe y emite flujos de datos a las entidades externas. Un modelo de implementación debe indicar qué dispositivos se utilizan para estas entradas o salidas: interruptores y señales luminosas o estaciones de trabajo, p.ej.
- **La elección de los dispositivos de almacenamiento.** Discos duros, discos ópticos, cintas magnéticas, etc.
- **El formato de las entradas y salidas.** Incluyendo aquí el tamaño de los datos (p.ej. longitud de los nombres, formato de fechas y números de teléfono, número de líneas máximo en un pedido, etc.). Todas estas restricciones deben ser incorporadas al diccionario de datos.

- **La secuencia de operaciones de entrada y salida.** Incluyendo la definición de cómo se van a hacer los diálogos del sistema con el usuario (orden de las pantallas de captura de datos y disposición de los datos en la pantalla, valores por defecto de los datos, mensajes de error y de ayuda, cómo se pasa de una pantalla a otra, etc.) Muchos de estos aspectos pueden definirse mediante DEs.
- **Volumen de datos.** El usuario debe indicar el volumen de datos que manejará el sistema de forma que se puedan prever las necesidades de almacenamiento y procesamiento.
- **Tiempo de respuesta.** El tiempo de respuesta de las aplicaciones interactivas y el tiempo máximo de procesamiento que puede consumir in proceso batch, etc.
- **Copias de seguridad y descarga de datos del sistema.** El usuario puede indicar cuándo y de qué datos es necesario hacer copia de seguridad o qué datos necesita mantener on-line (información del último año p.ej.) y qué datos se pueden descargar del sistema a cinta magnética. Habrá que prever mecanismos de recuperación o recarga de esa información cuando sea necesaria.
- **Seguridad.** Mecanismos de seguridad para evitar accesos no autorizados a todos o parte de los datos, a determinados procesos, etc.

5.6.- Ejemplos.

5.6.1.- Traductor y Distribuidor de Comunicaciones

Trataremos de modelizar un Traductor y Distribuidor de Comunicaciones (TDC) que sirva de interfaz entre un visualizador estándar y una serie de “suministradores de datos” a través de la red. La idea fundamental que subyace en este gestor es que los visualizadores de datos sean independientes del suministrador, de forma que si cambia el agente suministrador el desarrollo del visualizador siga siendo válido. Para ello suponemos que hay un protocolo perfectamente definido que describe los datos entre el visualizador y el gestor de comunicaciones, representación R1, y que el gestor de comunicaciones conoce también el protocolo de comunicación con el suministrador, representación R2,. La conversión de protocolos se encuentra almacenada en una base de datos, que denominaremos BD_POE. Para concretar más estas ideas pensemos en un sistema de visualización de datos de una sala de calderas de calefacción procedentes de un edificio inteligente (Figura 5.7).

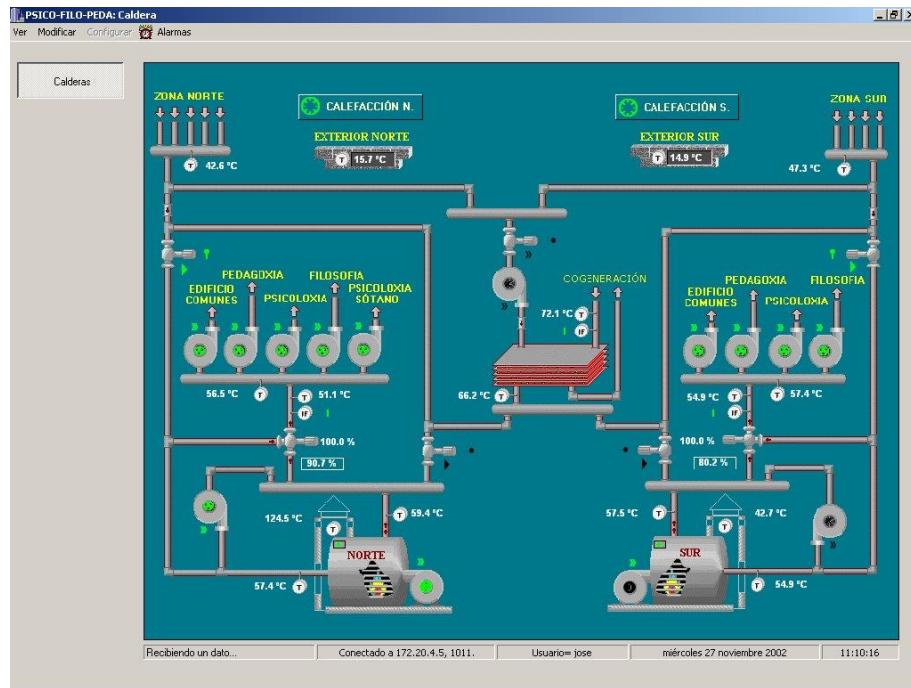


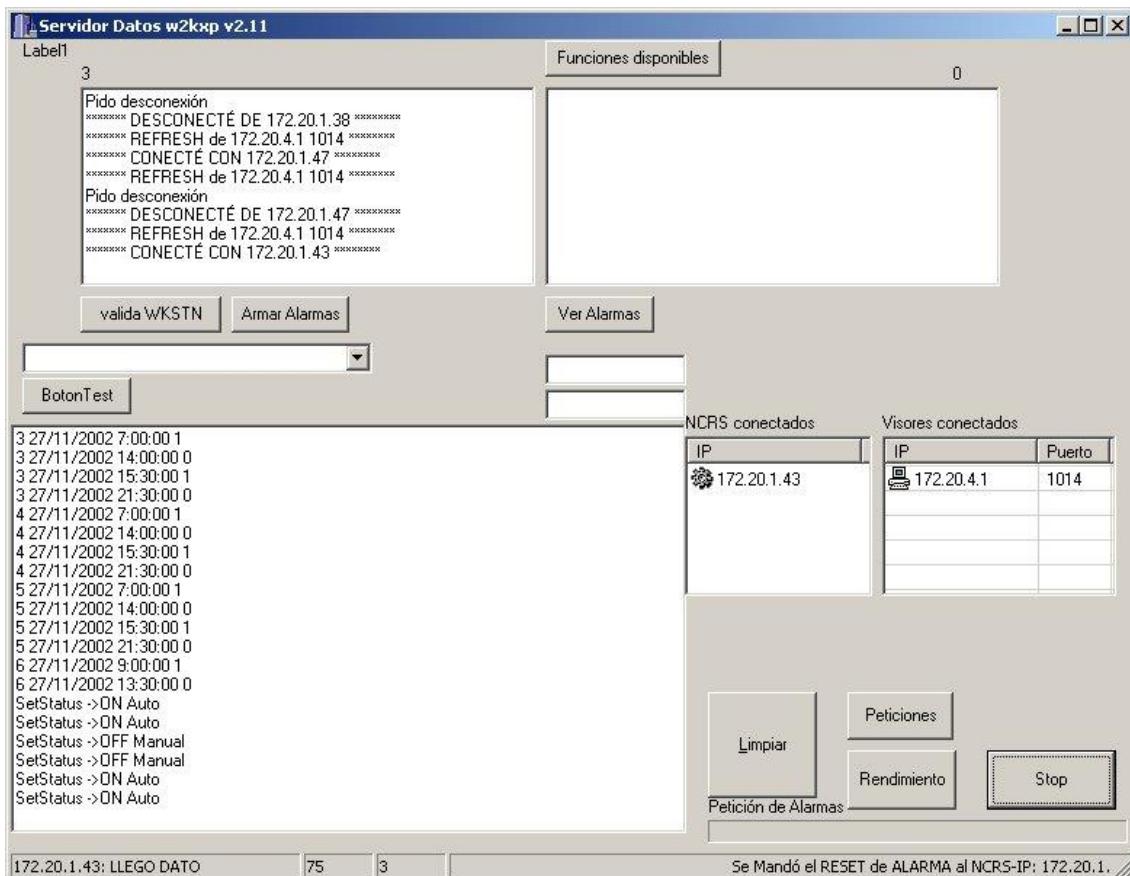
Figura 5.7.- Detalle del gráfico que representa a la sala de calderas

El suministrador es un ordenador que controla, supervisa y almacena todos los datos del edificio (datos de calefacción, agua y electricidad por ejemplo). Llamémosle NCRS, y su apariencia real es la podemos ver en la figura 5.8.



Figura 5.8.- Detalle del NCRS

Además el TDC está dotado de una ventana de monitorización que sirve, en un principio para que el desarrollador pueda monitorizar todo el tráfico de comunicaciones, tanto entre el TDC y los NCRS, como entre el TDC y los visualizadores. Esta ventana tendría el aspecto de la figura 5.9.

Figura 5.9.- Detalle del *Display* de monitorización

Estos tres elementos que acabamos de describir son para nuestro sistema agentes externos que consumen o producen datos. En el diagrama de contexto de nuestro gestor de comunicaciones, figura 5.10., situaremos al NCRS a la izda. y a la derecha indicando que es tanto productor como consumidor de datos.

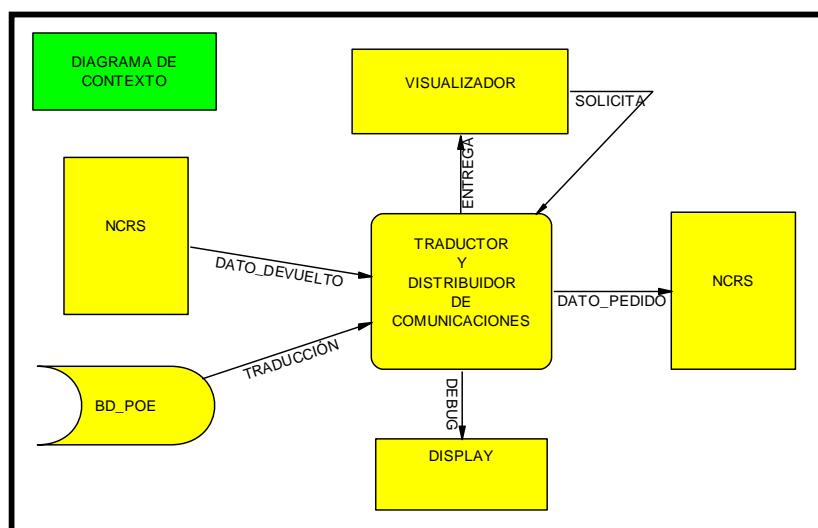


Fig. 5.10.- Diagrama de contexto del Traductor y Distribuidor de Comunicaciones

La BD_POE es un productor de datos, por lo que estará situada a la izda. El visualizador constituye una interfaz de usuario, mientras que el display, lo podemos entender como un interfaz de mantenimiento.

El DFD de nivel 1 describe los módulos principales del TDC, y su interacción con los agentes externos propuestos. Debemos tener presente que la comunicación con los visualizadores se realiza en la representación de nuestro sistema R1, mientras que la comunicación con los controladores de los edificios se hace en el protocolo de estos, esto es R2. En caso de tener varios suministradores de control en los edificios sería preciso replicar el módulo 3. Aunque como vimos no es habitual, en este primer nivel descomponemos el flujo de *debug*, que en el diagrama de contexto vimos como una interface con la pantalla en el debug de las comunicaciones en R1, DEBUG_CR1, y el de las comunicaciones en R2, DEBUG_CR2.

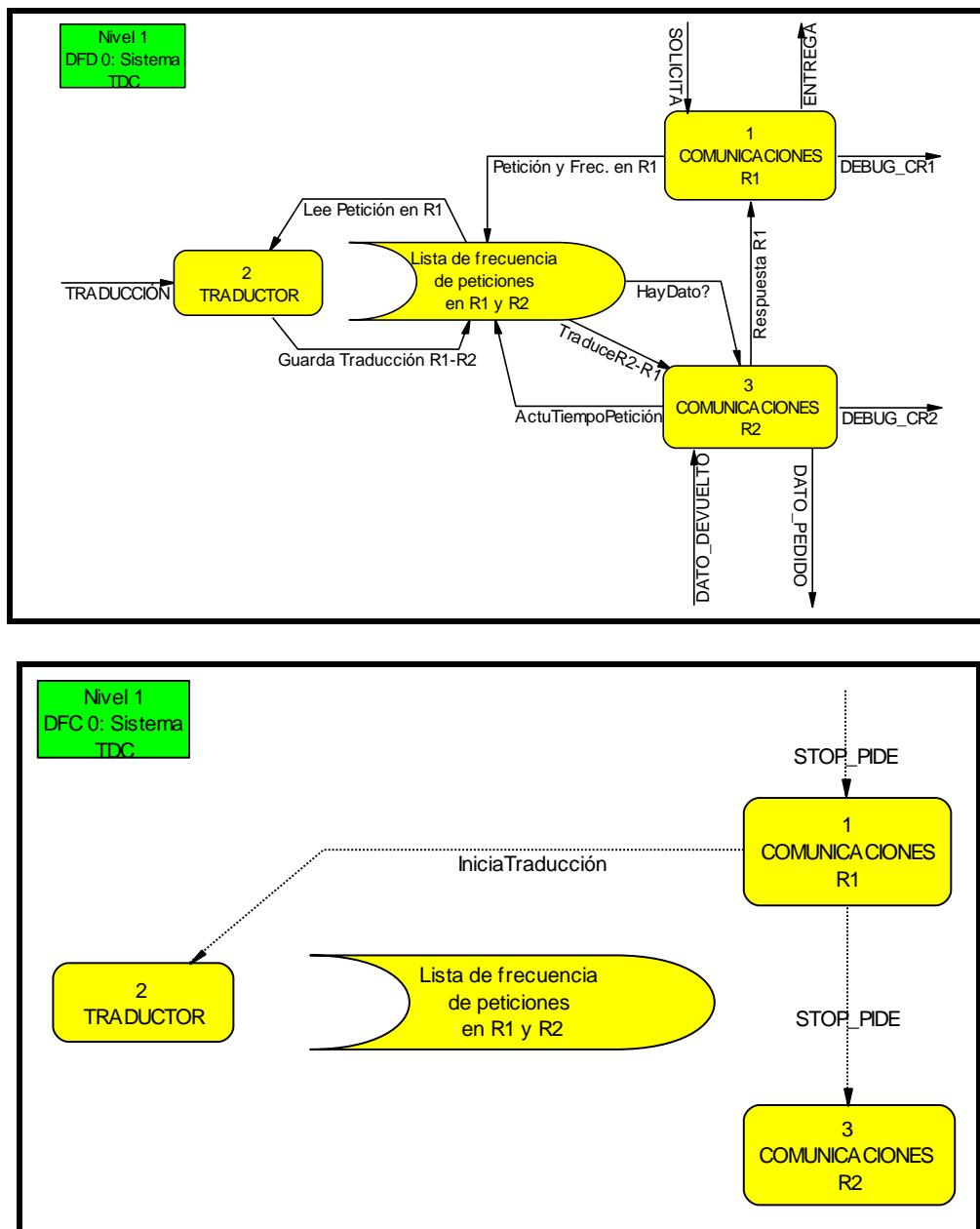


Figura 5.11.- DFD y DFC de nivel 1 del TDC

La figura 5.12 nos muestra el DFD que resulta de explotar el proceso 3. El diagrama de flujo de control (figura 5.13) nos muestra cuales son las señales y condiciones de datos que regulan el funcionamiento del TDC.

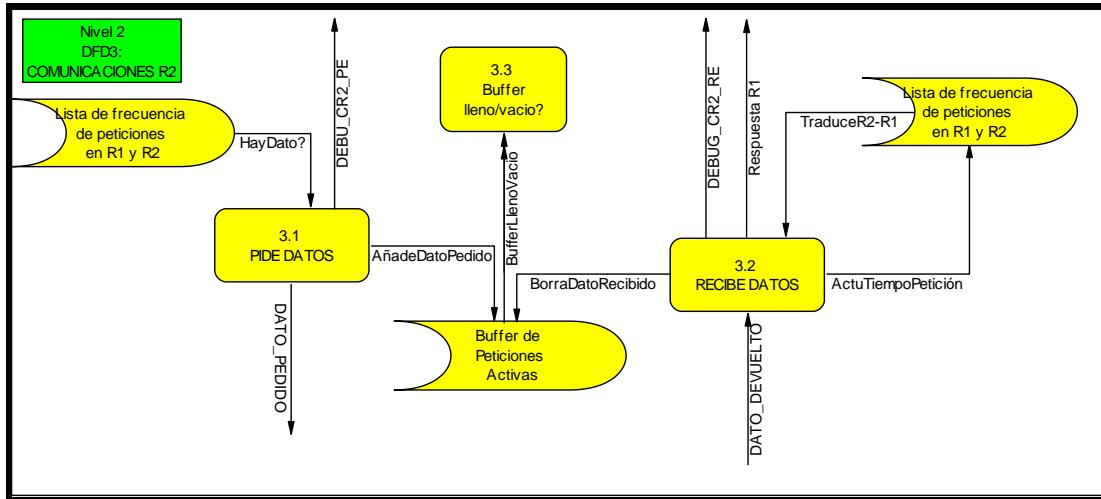


Figura 5.12.- DFD del proceso 3

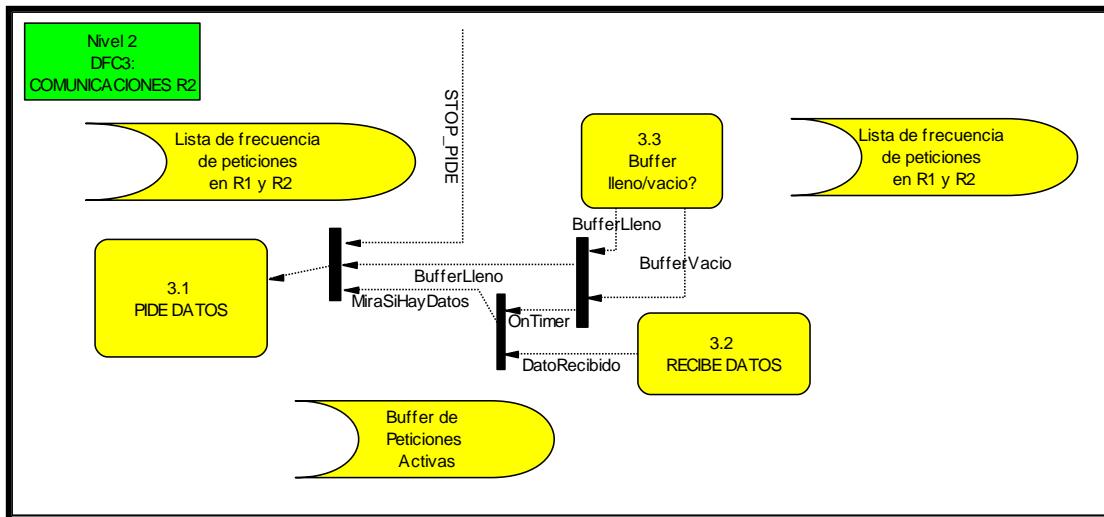


Figura 5.13.- DFC de nivel 1 del TDC

Las siguientes tablas de activación y desactivación forman parte de las Especificaciones de Control del DFC de nivel 2, junto con el Diagrama de Transición de Estados de la figura 5.14 y con la descripción natural de la figura 5.15, y nos muestran qué procesos se activan (o desactiva) bajo qué señales de control

	MiraSiHayDatos
3.1	1
3.2	0
3.3	0

Tabla de Activación

	STOP_PIDE	BufferLleno
3.1	1	1
3.2	0	0
3.3	0	0

Tabla de Desactivación

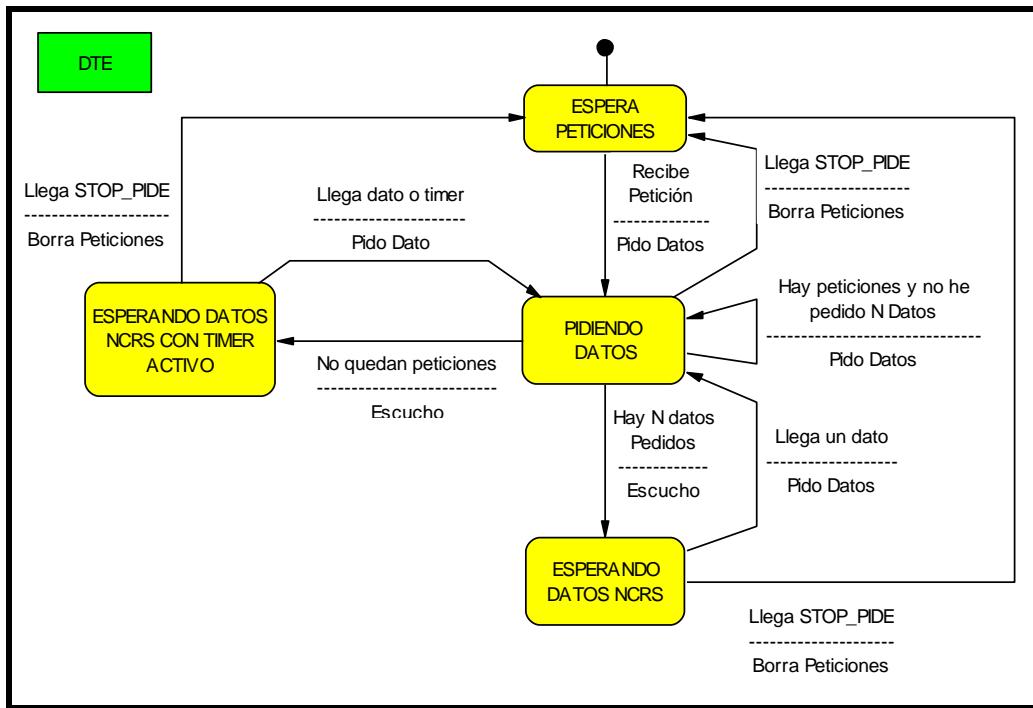


Figura 5.14. Diagrama de Transición de Estados del TDC

□ Descripción lenguaje estructurado

➤ ACTIVACIÓN DE *MiraSiHayDatos*

SI OnTimer **O** DatoRecibido

MiraSiHayDatos

➤ ACTIVACIÓN DE *OnTimer*

SI BufferVacio

MIENTRAS NO BufferLleno

ESPERA Retardo

OnTimer

Figura 5.15.- Descripción natural de la activación de los eventos MiraSiHayDatos y OnTimer

5.6.2.- Hogar Seguro

Para describir la construcción del modelado de este ejemplo vamos a suponer que somos un equipo interdisciplinar y que vamos a seguir una técnica de obtención de requisitos de *brainstorming*, denominada TFEA (Técnicas para Facilitar las Especificaciones de una Aplicación), tal y como vimos en el tema anterior. Para obtener una lista inicial de objetos vamos a utilizar el *Análisis Gramatical*, de la narrativa de procesamiento⁴ de forma que identificaremos los verbos con procesos y todos los nombres o expresiones sustantivadas, bien son entidades externas, bien flujos de datos o bien almacenes de datos.

Del análisis grammatical podemos hacer las siguientes listas:

Lista de objetos (Sustantivos):

- Panel de Control (Teclado, Display)
- Sensores (Humos, Aperturas, Movimiento)
- Alarma visual/sonora

⁴ La narrativa de procesamiento del software *Hogar Seguro* la podemos encontrar en Pressman5, pag. 212.

- Línea telefónica
- Usuario
- Contraseña
- Lista de Teléfonos
- Suceso
- Retardo
- Servicio de Monitorización
- Información relevante a un suceso

Lista de Servicios/Operaciones (verbos del AG):

- Permitir (Configurar/interactuar/programar)
- Supervisar (Sensores)
- Activar/Desactivar (Sistema)
- Contactar (Servicio de monitorización)
- Detectar (Suceso)
- Invocar (Alarma)
- Especificar (Retardo)
- Proporcionar (Información al Servicio Monitorización)
- Establecer (Llamada)
- Remarcar

Además, tras la supuesta reunión del equipo TFEA, obtendríamos las siguientes listas de rendimiento y restricciones:

Lista de Rendimiento:

- Respuesta inferior al segundo

Lista de Restricciones:

- Coste de fabricación < 200 €
 - Interfaz usable
 - Conexión a línea telefónica habitual
 - Conexión a línea telefónica inalámbrica
 - Funcionamiento mínimo independiente de electricidad
- El diagrama de contexto que generaríamos sería el que podemos ver en la figura 5.16.

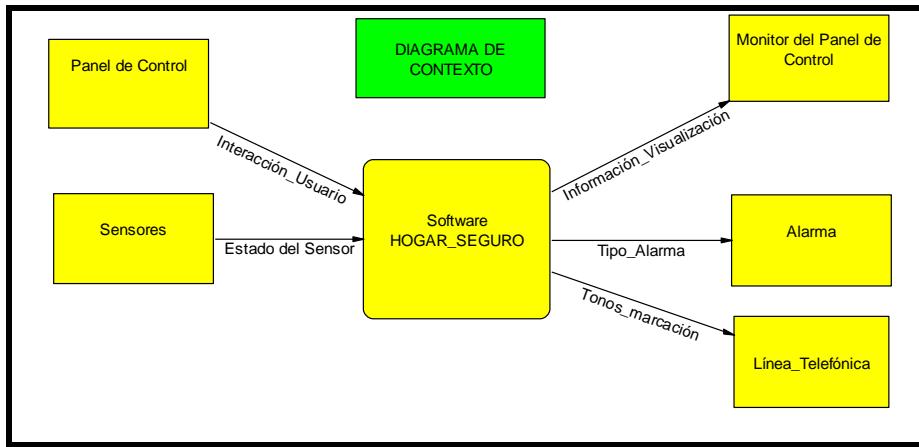


Figura 5.16.- Diagrama de contexto del software Hogar_Seguro

El diagrama de flujo de nivel 1, resultado de explotar el proceso número 0 que identifica al sistema como un todo en el diagrama de contexto, lo tenemos en la figura 5.17. De explotar el proceso 5, que hace referencia a la monitorización de los sensores y a la ejecución del procedimiento en caso de alarma, obtenemos la figura 5.18. Las siguientes (figs. 5.19 y 5.20) nos muestran el DFC y el DTE

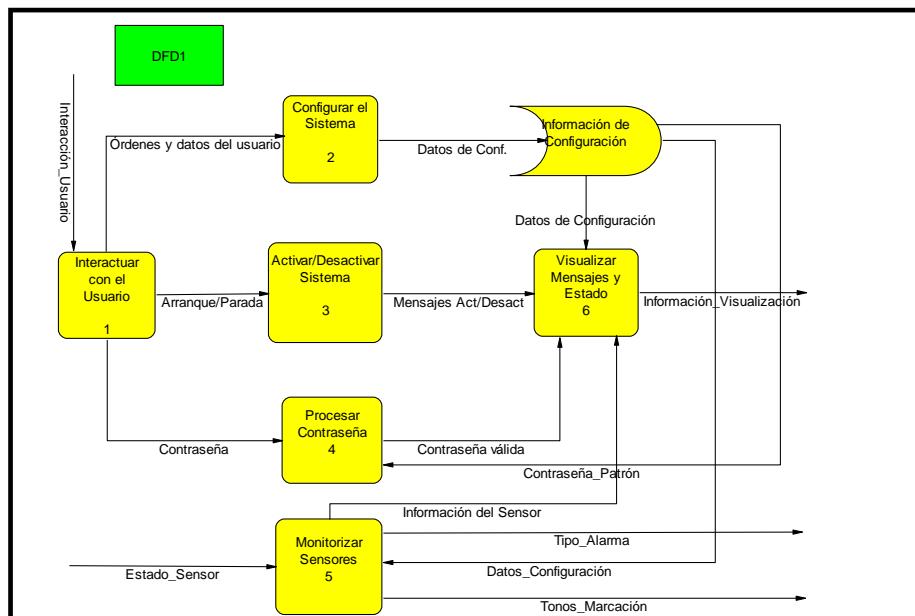


Figura 5.17.- DFD de nivel 1 del software Hogar_Seguro

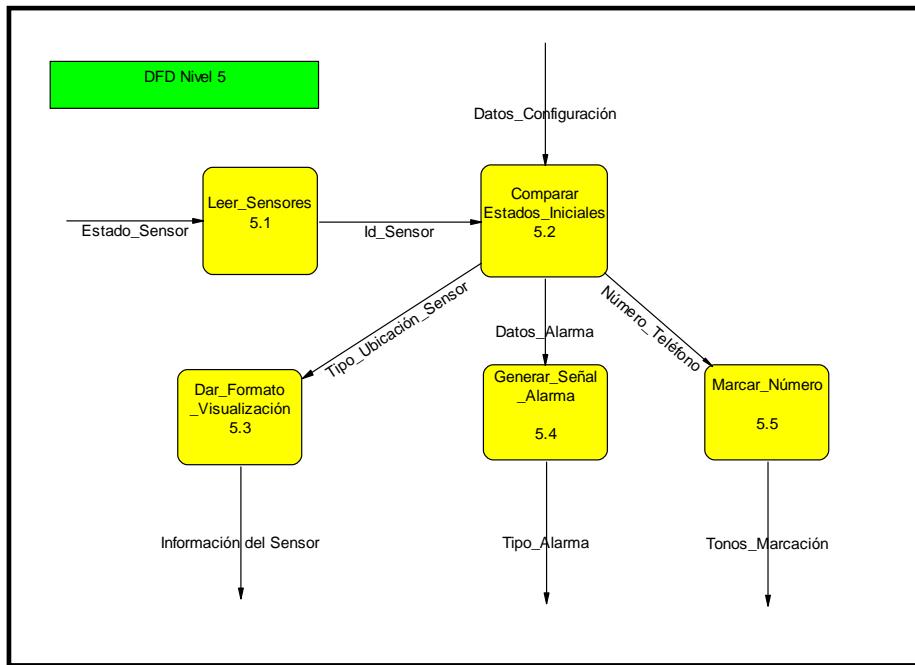


Figura 5.18.- DFD de nivel 5 del software Hogar_Seguro

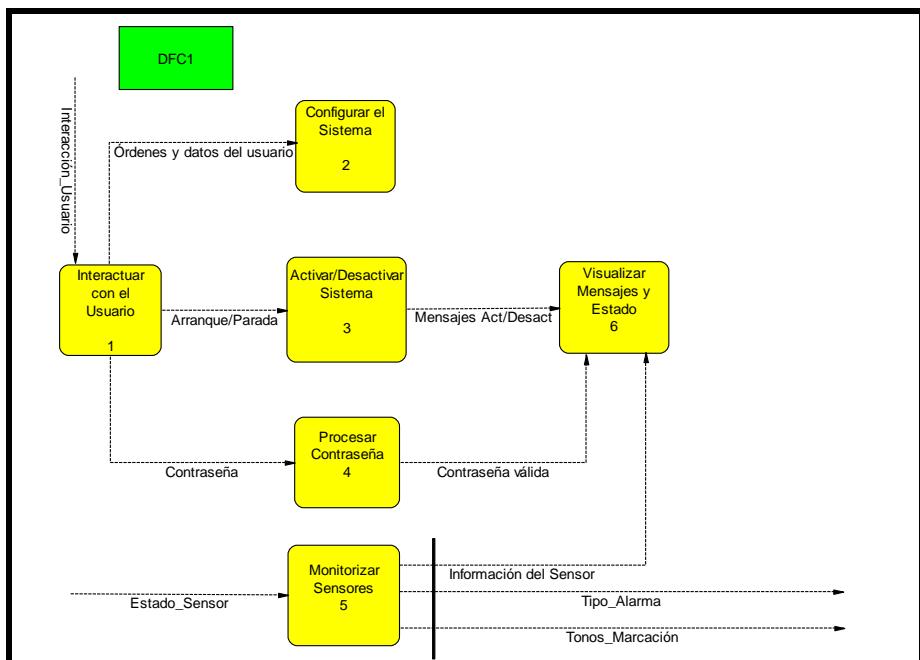


Figura 5.19.- Diagrama de flujo de control de nivel 1 del software Hogar_Seguro

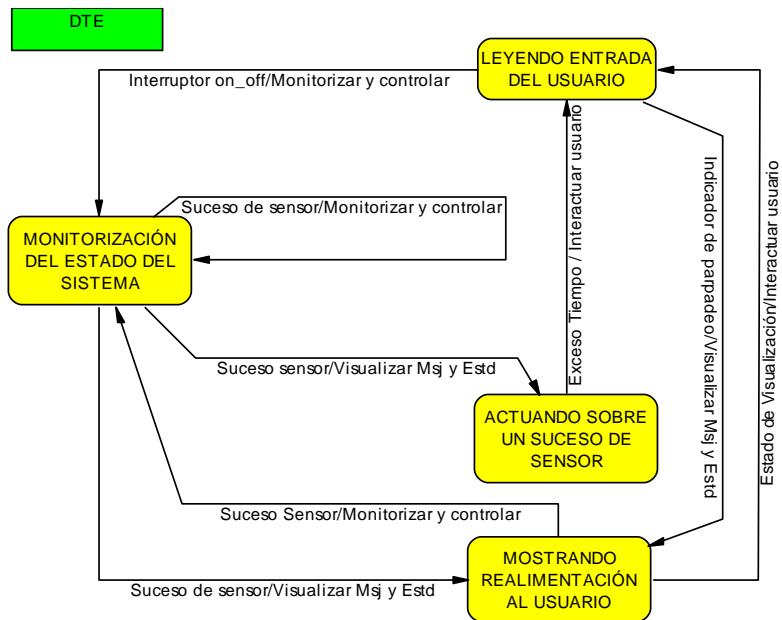


Figura 5.20.- Diagrama de estados del software Hogar_Seguro

Tema 6.- Pruebas del Software

Tema 6.- Pruebas del Software	163
6.1.- Introducción.....	165
6.1.1.- Definiciones.....	165
6.2.- Filosofía de las Pruebas del Software.....	166
6.3.- El Proceso de Prueba	169
6.4.- Técnicas de diseño de casos de prueba.....	170
6.5.- Pruebas Estructurales.....	171
6.5.1.- Utilización de la complejidad ciclomática de McCabe	175
6.6.- Prueba Funcional	178
6.6.1.- Particiones o clases de equivalencia	178
6.6.2.- Análisis de Valores Límite (AVL)	181
6.6.3.- Conjetura de errores	182
6.6.4.- Pruebas aleatorias	183
6.6.5.- Métodos de prueba basados en grafos	183
6.7.- enfoque práctico recomendado para el diseño de casos	186
6.8.- Documentación del diseño de las pruebas.....	187
6.8.1.- Plan de pruebas.....	189
6.8.2.- Especificación del diseño de pruebas	189
6.8.3.- Especificación de caso de prueba	190
6.8.4.- Especificación de procedimiento de prueba	190
6.9.- Ejecución de las pruebas	191
6.9.1.- El proceso de ejecución	191
6.9.2- Documentación de la ejecución de pruebas.....	192
6.9.3.- Histórico de pruebas	193
6.9.4.- Informe de incidente.....	193
6.9.5.- Informe resumen de las pruebas	193
6.9.6.- Depuración	194
6.9.7.- Análisis de errores a análisis causal	196
6.10.- Estrategia de aplicación de las pruebas	196
6.11.- Pruebas en desarrollos orientados a objetos	198

6.1.- Introducción.

Una de las características típicas del desarrollo de software basado en el ciclo de vida es la realización de controles periódicos, normalmente coincidiendo con los hitos de los proyectos o la terminación de documentos. Estos controles pretenden una evaluación de la calidad de los productos generados (especificación de requisitos, documentos de diseño, etc.) para poder detectar posibles defectos cuanto antes. Sin embargo, todo sistema o aplicación, independientemente de estas revisiones, debe ser probado mediante su ejecución controlada antes de ser entregado al cliente. Estas ejecuciones o ensayos de funcionamiento, posteriores a la terminación del código del software, se denominan habitualmente pruebas.

Las pruebas constituyen un método más (junto a las revisiones de los productos que preceden al código en el ciclo de vida) para poder verificar y validar el software. Se puede definir la verificación como: “El proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al principio de dicha fase”. Por ejemplo, verificar el código de un módulo significa comprobar si cumple lo marcado en la especificación de diseño donde se describe. Por otra parte, la validación es: “El proceso de evaluación del sistema o de uno de sus componentes durante o al final del desarrollo para determinar si satisface los requisitos especificados”. Así, validar una aplicación implica comprobar si satisface los requisitos marcados por el usuario. Podemos recurrir a la clásica explicación informal de Boehm de estos conceptos:

1. **Verificación:** ¿estamos construyendo correctamente el producto?
2. **Validación:** ¿estamos construyendo el producto correcto?

Como hemos dicho, las pruebas permiten verificar y validar el software cuando ya está en forma de código ejecutable. A continuación, expondremos algunas definiciones de conceptos relacionados con las pruebas.

6.1.1.- Definiciones

Las siguientes definiciones son algunas de las recogidas en el diccionario IEEE en relación a las pruebas [IEEE. 1990], que complementamos con otras más informales:

1. **Pruebas (test):** «una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto». Para Myers, probar (o la prueba) es el «proceso de ejecutar un programa con el fin de encontrar errores». El nombre «prueba», además de la actividad de probar, se puede utilizar para designar «un Conjunto de casos y procedimientos de prueba».
2. **Caso de prueba (test case):** «un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular como, por ejemplo, ejercitarse en un camino concreto de un programa o verificar el cumplimiento de un determinado requisito». También se puede referir a la documentación en la que se describen las entradas, condiciones y salidas de un caso de prueba.

3. **Defecto** (*defect, fault, «bug»*): «una incorrección en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa».
4. **Fallo** (*failure*): «La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados».
5. **Error** (*error*): tiene varias acepciones:
 1. La diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto. Por ejemplo, una diferencia de dos centímetros entre el valor calculado y el real.
 2. Un defecto. Por ejemplo, una instrucción incorrecta en un programa.
 3. Un resultado incorrecto. *Por ejemplo*, un programa ofrece como resultado de la raíz cuadrada de 36 el valor 7 en vez de 6.
 4. Una acción humana que conduce a un resultado incorrecto (una metedura de pata: *mistake*). Por ejemplo, que el operador o el programador pulse una tecla equivocada.

Nosotros desecharemos las acepciones 2 y 3, ya que coinciden con las definiciones de defecto y fallo, para evitar equívocos.

6.2.- Filosofía de las Pruebas del Software

Las especiales características del software (ausencia de existencia física, ausencia de leyes que rijan su comportamiento, gran complejidad, etc.) hacen aún más difícil la tarea de probarlo en relación con otros productos industriales. Como veremos posteriormente, la prueba exhaustiva del software es impracticable: no se pueden probar todas las posibilidades de su funcionamiento incluso en programas pequeños y sencillos. Además, existen prejuicios que pueden perjudicar a las pruebas. Todos estos factores obligan a estudiar cuál es la mejor actitud o filosofía a seguir en esta actividad.

Hay que recordar [MYERS, 1979], ante todo, que el objetivo de las pruebas es la detección de defectos en el software y que descubrir un defecto debería considerarse el éxito de una prueba. Se trata de una actividad a posteriori, de detección, y no de prevención, de problemas en el software. El problema es que, tradicionalmente, existe el mito de la ausencia de errores en el buen profesional: si fuéramos realmente capaces y empleáramos las técnicas más sofisticadas, no existirían los defectos en el software. Por lo tanto, un defecto implica que somos malos profesionales y que debemos sentirnos culpables. Si una prueba revela uno de estos problemas, implica la constatación de un fracaso del desarrollador.

Sin embargo, la realidad es muy distinta. Todo el mundo comete errores (*errare humanum est*) y es de sabios rectificar. Las pruebas permiten la rectificación en el software. La mayoría de los estudios revelan que los mejores programadores incluyen una cierta media de defectos por cada 1.000 líneas de código. Los defectos no son siempre el resultado de la negligencia, sino que en su aparición influyen múltiples factores (por ejemplo, la mala comunicación entre los miembros del equipo que da lugar a malentendidos en los requisitos pedidos; así, quizás sólo con telepatía no se cometan

errores). Por todo ello, el descubrimiento de un defecto significa un éxito para la mejora de la calidad al igual que, a pesar de lo incómodo que resulte, la detección de un problema de salud en un análisis médico se considera un éxito para lograr la curación del paciente.

Davis y Myers proponen una serie de recomendaciones para las pruebas. Davis propone¹:

1. A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente. Como hemos visto, el objetivo de las pruebas de software es descubrir errores. Se entiende que los defectos más graves (desde el punto de vista del cliente) son aquellos que impiden al programa cumplir sus requisitos.
2. Las pruebas deberían planificarse mucho antes de que empiecen. La planificación de las pruebas puede comenzar tan pronto como esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se ha consolidado. Por tanto, se pueden planificar y diseñar algunas pruebas antes de generar ningún código.
3. El 80% de los errores surgen al hacer un seguimiento del 20% de los módulos del software (Principio de Pareto). El problema, por supuesto consiste en aislar los módulos sospechosos y probarlos concienzudamente.
4. Las pruebas tendrían que hacerse de lo pequeño hacia lo grande. Las primeras pruebas planeadas y ejecutadas se centran generalmente en módulos individuales del programa. A medida que avanzan las pruebas, desplazan su punto de mira en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero.
5. No son posibles pruebas exhaustivas. El número de permutaciones de caminos para incluso un programa de tamaño moderado es excepcionalmente grande. Por este motivo, es imposible ejecutar todas las combinaciones de caminos durante las pruebas. Es posible, sin embargo, cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
6. Las pruebas, para ser más efectivas, deberían de ser realizadas por un equipo independiente del equipo de desarrollo del software. Por “más eficaces” queremos referirnos a pruebas con más alta probabilidad de encontrar errores (el objetivo principal de las pruebas). El programador debe evitar probar sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas. Esta actitud inadecuada lleva a realizar pruebas menos rigurosas de lo que sería deseable. Además, es normal que las situaciones que ha olvidado considerar al crear el programa (por ejemplo, no pensar en cómo tratar un fichero de entrada vacío) queden de nuevo olvidadas al escribir casos de prueba. Lo ideal sería que probara el software el peor

¹ Pressman 5^a Edición, pp282

enemigo de quien lo ha construido, ya que así se aseguraría una búsqueda implacable de cualquier error cometido.

Por su parte las recomendaciones de G. J. Myers para las pruebas son las siguientes:

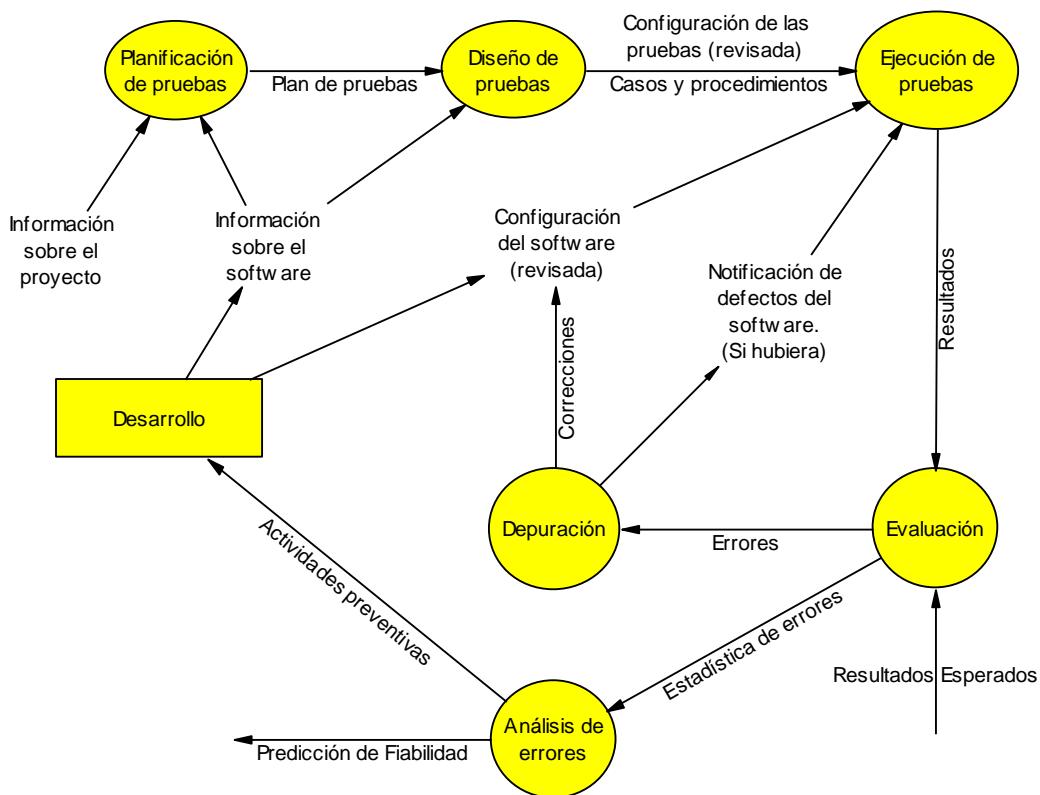
1. Cada caso de prueba debe definir el resultado de salida esperado. Este resultado esperado es el que se compara con el realmente obtenido de la ejecución en la prueba. Las discrepancias entre ambos (errores) se consideran síntomas de un posible defecto en el software.
2. Se debe inspeccionar a conciencia el resultado de cada prueba para así, poder descubrir posibles síntomas de defectos. Lamentablemente, es frecuente pasar por alto síntomas bastante claros. Esto invalida todo el esfuerzo realizado en la planificación, diseño y ejecución de pruebas.
3. Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados. Es frecuente observar una tendencia a centrarse en lo esperado y lo válido.
4. Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):
 - Probar si el software no hace lo que debe hacer.
 - Probar si el software hace lo que no debe hacer, es decir, si provoca efectos secundarios adversos,
5. Se deben evitar los casos desecharables, es decir, los no documentados ni diseñados con cuidado (por ejemplo, los que se teclean sobre la marcha), ya que suele ser necesario probar una y otra vez el software hasta que queda libre de defectos. No documentar o guardar los casos significa repetir constantemente el diseño de casos de prueba.
6. No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas. Hay que asumir que siempre hay defectos (ya están cuantificadas las tasas habituales de defectos de los mejores desarrolladores profesionales de software y no son cero) y que hay que detectarlos. Las estadísticas confirman que, prácticamente, el 40% del esfuerzo de desarrollo se consume en pruebas y depuración.
7. Las pruebas son una tarea tanto o más creativa que el desarrollo de software. Siempre se han considerado las pruebas como una tarea destructiva y rutinaria. No obstante, no existen técnicas rutinarias «como veremos para el diseño de pruebas y hay que recurrir al ingenio para alcanzar un buen nivel de detección de defectos con los recursos disponibles. Myers habla en su famoso libro [MYERS. 1979] del «arte de las pruebas». También se suele decir que «si cree que la construcción del programa ha sido difícil, aún no ha visto nada».

A los principios de Myers también es preciso añadir el principio de pareto y de que el grupo de pruebas tiene que ser independiente del de programación que ya recogíamos con Davis y que Myers repite.

Por lo tanto, la filosofía más adecuada para las pruebas consiste en planificarlas y diseñarlas de forma sistemática para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo. Así, debemos recordar que «un buen caso de prueba es aquel que tiene una gran probabilidad de encontrar un defecto no descubierto aún» y que «el éxito de una prueba consiste en detectar un defecto no encontrado antes». El contraste con la visión tradicional de las pruebas («vamos a probar un par de opciones para comprobar que funciona y ya está») es radical.

6.3.- El Proceso de Prueba

En la Figura se puede ver una representación del proceso completo relacionado con las pruebas basada en parte en el estándar y en parte en Pressman.



El proceso de prueba comienza con la generación de un plan de pruebas en base a la documentación sobre el proyecto y la documentación sobre el software a probar. A partir de dicho plan se entra en detalle diseñando pruebas específicas basándose en la documentación del software a probar. Una vez detalladas las pruebas (especificaciones de casos y de procedimientos) se toma la configuración del software (revisada, para confirmar que se trata de la versión apropiada del programa) que se va a probar para ejecutar sobre ella los casos. En algunas situaciones, se puede tratar de reejecuciones de pruebas, por lo que es conveniente tener constancia de los defectos ya detectados aunque aún no corregidos. A partir de los resultados de salida, se pasa a su evaluación mediante comparación con la salida esperada. A partir de ésta, se pueden realizar dos actividades:

1. La depuración (localización y corrección de defectos).
2. El análisis de la estadística de errores.

La depuración puede corregir o no los defectos. Si no consigue localizarlos, puede ser necesario realizar pruebas adicionales para obtener más información. Si se corrige un defecto, se debe volver a probar el software para comprobar que el problema está resuelto.

Por su parte, el análisis de errores puede servir para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y mejorar los procesos de desarrollo.

6.4.- Técnicas de diseño de casos de prueba

El diseño de casos de prueba está totalmente mediatisado por la imposibilidad de probar exhaustivamente el software. Pensemos en un programa muy sencillo que sólo suma dos números enteros de dos cifras (del 0 al 99). Si quisieramos probar, simplemente, todos los valores válidos que se pueden sumar, deberíamos probar 10.000 combinaciones distintas de cien posibles números del primer sumando y cien del segundo. Pensemos que aún quedarían por probar todas las posibilidades de error al introducir los datos (por ejemplo, teclear una letra en vez de un número). La conclusión es que, si para un programa tan elemental debemos realizar tantas pruebas, pensemos en lo que supondría un programa medio.

En consecuencia, las técnicas de diseño de casos de prueba tienen como objetivo conseguir una confianza aceptable en que se detectarán los defectos existentes (ya que la seguridad total sólo puede obtenerse de la prueba exhaustiva, que no es practicable) sin necesidad de consumir una cantidad excesiva de recursos (por ejemplo, tiempo para probar o tiempo de ejecución). Toda la disciplina de pruebas debe moverse, por lo tanto, en un equilibrio entre la disponibilidad de recursos y la confianza que aportan los casos para descubrir los defectos existentes. Este equilibrio no es sencillo, lo que convierte a las pruebas en una disciplina difícil que está lejos de parecerse a la imagen de actividad rutinaria que suele sugerir.

Ya que no se pueden probar todas las posibilidades de funcionamiento del software, la idea fundamental para el diseño de casos de prueba consiste en elegir algunas de ellas que, por sus características, se consideren representativas del resto. Así, se asume que, si no se detectan defectos en el software al ejecutar dichos casos, podemos tener un cierto nivel de confianza (que depende de la elección de los casos) en que el programa no tiene defectos. La dificultad de esta idea es saber elegir los casos que se deben ejecutar. De hecho, una elección puramente aleatoria no proporciona demasiada confianza en que se puedan detectar todos los defectos existentes. Por ejemplo, en el caso del programa de suma de dos números, elegir cuatro pares de sumandos al azar no aporta mucha seguridad a la prueba (una probabilidad de 4/10000 de cobertura de posibilidades). Por eso es necesario recurrir a ciertos criterios de elección que veremos a continuación.

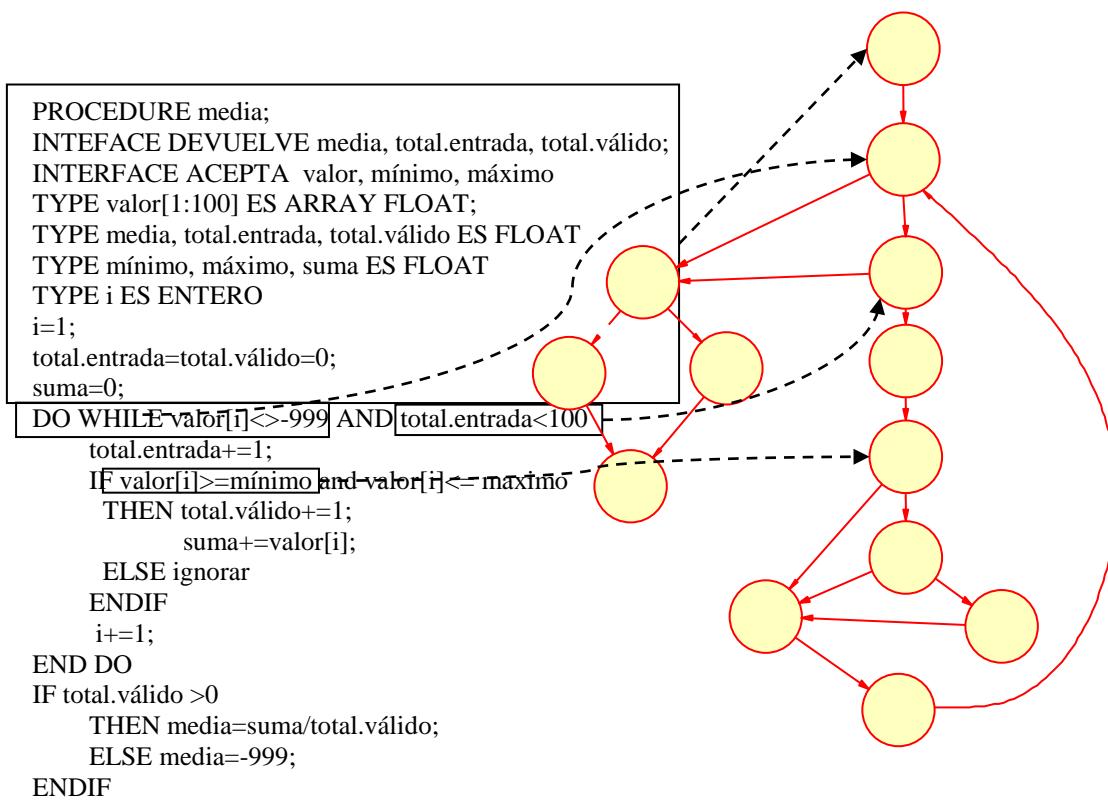
Existen dos enfoques principales para el diseño de casos:

1. El enfoque estructural o de caja blanca. Consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. En este caso, la prueba ideal (exhaustiva) del software consistiría en probar todos los posibles caminos de ejecución, a través de las instrucciones del código, que puedan trazarse.

2. El enfoque funcional o de caja negra. Consiste en estudiar la especificación de las funciones, la entrada y la salida, para derivar los casos. Aquí, la prueba ideal del software consistiría en probar todas las posibles entradas y salidas del programa. La prueba de entradas y salidas supone la búsqueda de casos de prueba que admite dos posibilidades
- Búsqueda estructurada: que propone la búsqueda sistemática de un reducido número de pruebas que sean representativas del mayor número posible de situaciones de funcionamiento del sistema
 - Búsqueda aleatoria que consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba. La prueba exhaustiva consistiría en probar todas las posibles entradas al programa.

Estos enfoques no son excluyentes entre sí, ya que se pueden combinar para conseguir una detección de defectos más eficaz. Veremos a continuación Una presentación de cada uno de ellos.

6.5.- Pruebas Estructurales

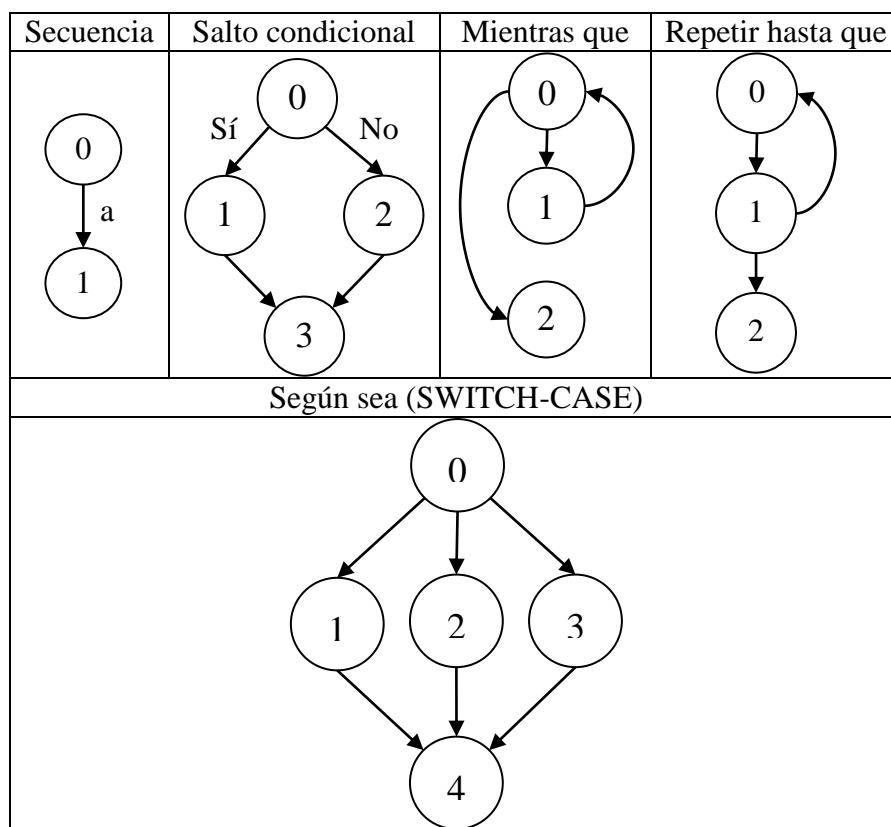


Como hemos dicho, las pruebas exhaustivas son impracticables. Podemos recurrir al clásico ejemplo de Myers de un programa de 50 líneas con 25 sentencias IF en serie, en el que el número total de caminos contiene 33,5 millones de secuencias potenciales (contando dos posibles salidas para cada IF tenemos 2^{25} posibles caminos). El diseño de casos tiene que basarse en la elección de caminos importantes que ofrezcan una seguridad aceptable de descubrir un defecto, y para ello se utilizan los llamados criterios de cobertura lógica. Antes de pasar a examinarlos, conviene señalar que estas técnicas no requieren el uso de ninguna representación gráfica específica del software, aunque es

habitual tomar como base los llamados diagramas de flujo de control (*flowgraph charts flowcharts*). Como ejemplo de diagrama de flujo junto al código correspondiente podemos ver la figura siguiente

Para dibujar el grafo de flujo de un programa es recomendable seguir los siguientes pasos:

1. Señalar sobre el código cada condición de cada decisión (por ejemplo, en la figura `valor[i]>-999` y `total.entrada<100` son condiciones distintas del primer bucle) tanto en sentencias IF-THEN y SWITCH-CASE como en los bucles FOR, DO o WHILE.
2. Agrupar el resto de las sentencias en secuencias situadas entre cada dos condiciones según los esquemas de representación de las estructuras básicas que mostramos a continuación.



3. Numerar tanto las condiciones como los grupos de sentencias, de manera que se les asigne un identificador único. Es recomendable alterar el orden en el que aparecen las condiciones en una decisión multicondicional, situándolas en orden decreciente de restricción (primero, las más restrictivas). El objetivo de esta alteración es facilitar la derivación de casos de prueba una vez obtenido el grafo. Es conveniente identificar los nodos que representan condiciones asignándoles una letra y señalar cuál es el resultado que provoca la ejecución de cada una de las aristas que surgen de ellos (por ejemplo, si la condición x es verdadera o falsa).

Una posible clasificación de criterios de cobertura lógica es la que se ofrece abajo (MYERS, 1979). Hay que destacar que los criterios de cobertura que se ofrecen están en orden de exigencia y, por lo tanto, de coste económico. Es decir, el criterio de cobertura de sentencias es el que ofrece una menor seguridad de detección de defectos, pero es el que cuesta menos en número de ejecuciones del programa.

1. **Cobertura de sentencias.** Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.
2. **Cobertura de decisiones.** Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso. En general, una ejecución de pruebas que cumple la cobertura de decisiones cumple también la cobertura de sentencias.
3. **Cobertura de condiciones.** Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte su valor verdadero al menos una vez y el falso al menos una vez. No podemos asegurar que si se cumple la cobertura de condiciones se cumple necesariamente la de decisiones.
4. **Criterio de decisión/condición.** Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.
5. **Criterio de condición múltiple.** En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea (por ejemplo, según se ejecuta en el procesador se podría considerar que cada decisión multicondicional se descompone en varias decisiones unicondicionales). Es decir, una decisión como IF (a=1) AND (c=4) THEN se convierte en una concatenación de dos decisiones: IF (a=1) y IF (c=4). En este caso, debemos conseguir que todas las combinaciones posibles de resultados (verdadero/falso) de cada condición en cada decisión se ejecuten al menos una vez.

La cobertura de caminos (secuencias de sentencias) es el criterio más elevado: cada uno de los posibles caminos del programa se debe ejecutar al menos una vez. Se define camino como la secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final. Como hemos visto, el número de caminos en un programa pequeño puede ser impracticable para las pruebas. Para reducir el número de caminos a probar, se habla del concepto de camino de prueba (*test path*): un camino del programa que atraviesa, como máximo, una vez el interior de cada bucle que encuentra. La idea en la que se basa consiste en que ejecutar un bucle más de una vez no supone una mayor seguridad de detectar defectos en él. Sin embargo, otros especialistas [HUANG, 1979] recomiendan que se pruebe cada bucle tres veces: una sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces. Esto último es interesante para comprobar cómo se comporta a partir de los valores de datos procedentes, no del exterior del bucle (como en la primera iteración), sino de las operaciones de su interior. Saber cuál es el número de caminos del grafo de un programa ayuda a planificar las pruebas y a asignar recursos a las mismas, ya que indica el número de ejecuciones necesarias. También sirve de comprobación a la hora de enumerar los caminos.

Los bucles constituyen el elemento de los programas que genera un mayor número de problemas para la cobertura de caminos. Su tratamiento no es sencillo ni siquiera adoptando el concepto de camino de prueba. Pensemos, por ejemplo, en el caso de varios bucles anidados o bucles que fijan valores mínimo y máximo de repeticiones.

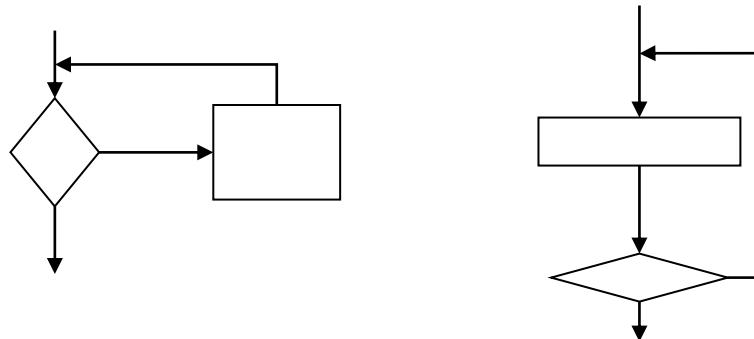
Prueba de bucles

Los bucles son la piedra angular de la inmensa mayoría de los algoritmos implementados en software. Y sin embargo, les prestamos normalmente poca atención cuando llevamos a cabo las pruebas del software.

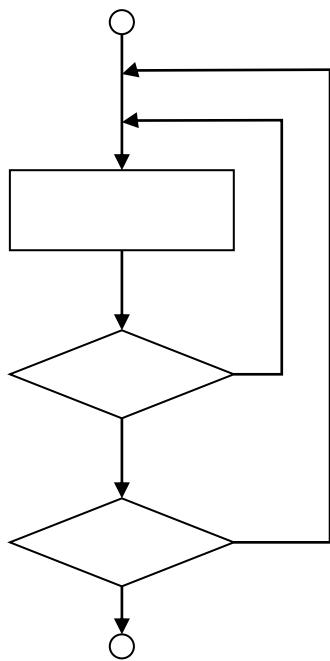
La prueba de bucles es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles. Se pueden definir cuatro clases diferentes de bucles: bucles simples, bucles concatenados, bucles anidados y bucles no estructurados.

Bucles simples. A los bucles simples se les debe aplicar el siguiente conjunto de pruebas, donde n es el número máximo de pasos permitidos por el bucle:

1. pasar por alto totalmente el bucle
2. pasar una sola vez por el bucle
3. pasar dos veces por el bucle
4. hacer m pasos por el bucle con $m < n$
5. hacer $n - 1$ y $n + 1$ pasos por el bucle



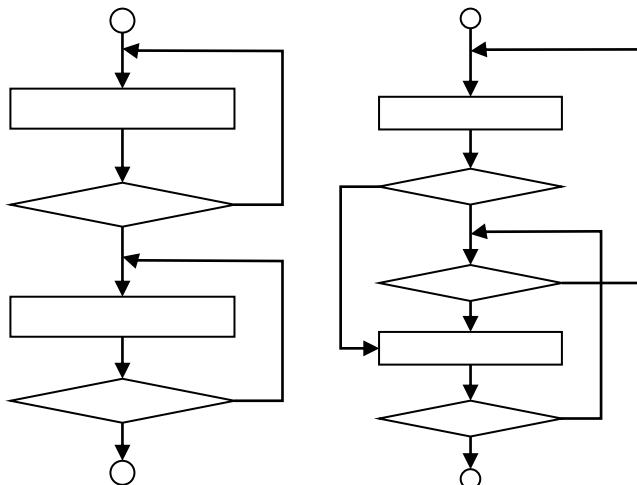
Bucles anidados. Si extendiéramos el enfoque de prueba de los bucles simples a los bucles anidados, el número de posibles pruebas aumentaría geométricamente a medida que aumenta el nivel de anidamiento. Esto llevaría a un número impracticable de pruebas. Beizer [BEI90] sugiere un enfoque que ayuda a reducir el número de pruebas:



1. Comenzar por el bucle más interior. Establecer o configurar los demás bucles con sus valores mínimos.
2. Llevar a cabo las pruebas de bucles simples para el bucle más interior mientras se mantienen los parámetros de iteración (por ejemplo, contador del bucle) de los bucles externos en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.
3. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores «típicos».
4. Continuar hasta que se hayan probado todos los bucles.

Bucles concatenados. Los bucles concatenados se pueden probar mediante el enfoque anteriormente definido para los bucles simples, mientras cada uno de los bucles sea independiente del resto. Sin embargo, si hay dos bucles concatenados y se usa el controlador del bucle 1 como valor inicial del bucle 2, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles se deben *rediseñar* para que se ajusten a las construcciones de programación estructurada



Bucle Concatenados

Bucle no estructurados

6.5.1.- Utilización de la complejidad ciclomática de McCabe

La utilización de la métrica de McCabe ha sido muy popular en el diseño de pruebas desde su creación. Esta métrica es un indicador del número de caminos independientes que existen en un grafo. El propio McCabe definió como un buen criterio de prueba la

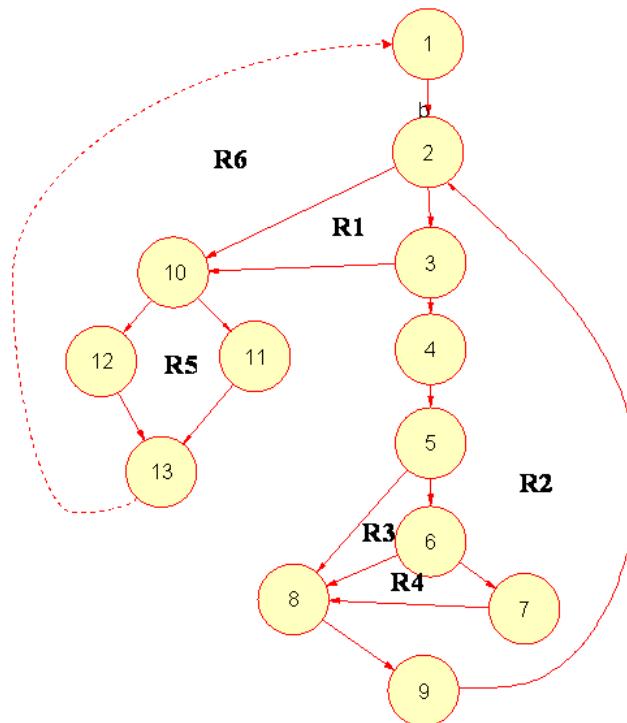
consecución de la ejecución de un conjunto de caminos independientes, lo que implica probar un número de caminos igual al de la métrica. Se propone este criterio como equivalente a una cobertura de decisiones, aunque se han propuesto contraejemplos que invalidan esta suposición.

McCabe habla de caminos linealmente independientes como una idea basada en los conceptos matemáticos de base de vectores y vectores linealmente independientes. En la práctica, puede decirse que un camino es independiente de otros si incorpora un arco o arista del grafo que los demás no incluyen, aunque ésta no es una equivalencia rigurosa. La métrica de McCabe coincide con el número máximo de caminos independientes que puede haber en un grafo.

La complejidad de McCabe $V(G)$ se puede calcular de las tres maneras siguientes a partir de un grafo de flujo G :

1. $V(G) = a - n + 2$, siendo a el número de arcos o aristas del grafo y n número de nodos.
 2. $V(G) = r$, siendo r el número de regiones cerradas del grafo.
 3. $V(G) = c + 1$, siendo c el número de nodos de condición.

Veamos cómo se aplican estas fórmulas sobre el grafo de flujo de la figura:



1. $V(G) = a-n+2 = 17-13+2 = 6$. Para facilitar esta tarea podemos marcar los arcos y los nodos con un número, en nuestro ejemplo sólo hemos identificado los nodos.
 2. $V(G) = 6$. Las regiones o áreas cerradas (limitadas por aristas) del grafo son seis. Las hemos marcado en el grafo con **Rn**. Como puede verse, se ha

marcado un área región 6 añadiendo un arco ficticio, discontinuo, desde el nodo 13, nodo de salida, al nodo 1. Esto se debe a que las fórmulas de McCabe sólo son aplicables a grafos fuertemente conexos, es decir, aquellos para los cuales existe un camino entre cualesquiera dos nodos que se elijan. Los programas, con un nodo de inicio y otro de final, no cumplen ésta condición. Por eso, debemos marcar dicho arco o, como alternativa, contabilizar la región externa al grafo como una más.

3. $V(G) = P+I=5+1=6$. Los nodos de condición son el 2, el 3, el 5, el 6 y el 10. Todos ellos son nodos de decisión binaria, es decir, surgen dos aristas de ellos. En el caso de que de un nodo de condición (por ejemplo, una sentencia Case-of) partieran n arcos ($n > 2$), debería contabilizarse como $n-1$ para la fórmula (que equivale al número de bifurcaciones binarias necesarias para simular dicha bifurcación « n -aria»).

Una vez calculado el valor y $V(G)$ podemos afirmar que el número máximo de caminos independientes de dicho grafo es seis. El criterio de prueba de McCabe consiste en elegir seis caminos que sean independientes entre sí y crear casos de prueba cuya ejecución siga dichos caminos. Para ayudar a la elección de dichos caminos, McCabe creó un procedimiento llamado «método del camino básico», consistente en realizar variaciones sobre la elección de un primer camino de prueba típico denominado camino básico.

Los caminos básicos seleccionados en nuestro caso, descritos como secuencias de nodos, serían los siguientes:

1. 1-2-10-11-13
2. 1-2-10-12-13
3. 1-2-3-10-11-13
4. 1-2-3-4-5-8-9-2-10-11-13
5. 1-2-3-4-5-6-8-9-2-10-11-13
6. 1-2-3-4-5-6-7-8-9-2-10-11-13

Hemos subrayado los elementos de cada camino que lo hacen independiente de los demás. Conviene aclarar que algunos de los caminos quizás no se puedan ejecutar solos y requieran una concatenación con algún otro. A partir de estos caminos, el diseñador de las pruebas debe analizar el código para saber los datos de entrada necesarios para forzar la ejecución de cada uno de ellos. Una vez determinados los datos de entrada hay que consultar la especificación para averiguar cuál es la salida teóricamente correcta para cada caso.

Puede ocurrir también que las condiciones necesarias para que la ejecución pase por un determinado camino no se puedan satisfacer de ninguna manera. Nos encontraríamos entonces ante un «camino imposible». En ese caso, debemos sustituir dicho camino por otro posible que permita satisfacer igualmente el criterio de prueba de McCabe, es decir, que ejecute la misma arista o flecha que diferencia al imposible de los demás caminos independientes.

La experimentación con la métrica de McCabe ha dado como resultado las siguientes conclusiones:

- $V(G)$ marca un límite mínimo de número de casos de prueba para un programa, contando siempre cada condición de decisión como un nodo individual.
- Parece que cuando $V(G)$ es mayor que diez la probabilidad de defectos en el módulo o en el programa crece bastante si dicho valor alto no se debe a sentencias Case-of o similares. En estos casos, es recomendable replantearse el diseño modular obtenido, dividiendo los módulos para no superar el límite de diez de la métrica de McCabe en cada uno de ellos.

6.6.- Prueba Funcional

La prueba funcional o de caja negra se centra en el estudio de la especificación del software, del análisis de las funciones que debe realizar y de las entradas y de las salidas. Lamentablemente, la prueba exhaustiva de caja negra también es generalmente impracticable: pensemos en el ejemplo de la suma visto anteriormente. De nuevo, ya que no podemos ejecutar todas las posibilidades de funcionamiento y todas las combinaciones de entradas y de salidas, debemos buscar criterios que permitan elegir un subconjunto de casos cuya ejecución aporte una cierta confianza en detectar los posibles defectos del software. Para fijar estas pautas de diseño de pruebas, nos apoyaremos en las siguientes dos definiciones de Myers que definen qué es un caso de prueba bien elegido:

- El que reduce el número de otros casos necesarios para que la prueba sea razonable. Esto implica que el caso ejecute el máximo número de posibilidades de entrada diferentes para así reducir el total de casos.
- Cubre un conjunto extenso de otros casos posibles, es decir, nos indica algo acerca de la ausencia o la presencia de defectos en el conjunto específico de entradas que prueba, así como de otros conjuntos similares.

Veremos a continuación distintas técnicas de diseño de casos de caja negra.

6.6.1.- Particiones o clases de equivalencia

Esta técnica utiliza las cualidades que definen un buen caso de prueba de la siguiente manera:

- Cada caso debe cubrir el máximo número de entradas.
- Debe tratarse el dominio de valores de entrada dividido en un número finito de clases de equivalencia que cumplan la siguiente propiedad: la prueba de un valor representativo de una clase permite suponer «razonablemente» que el resultado obtenido (existan defectos o no) será el mismo que el obtenido probando cualquier otro valor de la clase.

El método de diseño de casos consiste entonces en:

- Identificación de clases de equivalencia.
- Creación de los casos de prueba correspondientes.

Para identificar las posibles clases de equivalencia de un programa a partir de su especificación se deben seguir los siguientes pasos:

1. Identificación de las condiciones de las entradas del programa, es decir, restricciones de formato o contenido de los datos de entrada.
2. A partir de ellas, se identifican clases de equivalencia que pueden ser:
 - De datos válidos.
 - De datos no válidos o erróneos.

La identificación de las clases se realiza basándose en el principio de igualdad de tratamiento: todos los valores de la clase deben ser tratados de la misma manera por el programa.

3. Existen algunas reglas que ayudan a identificar clases:

- R1.- Si se especifica un rango de valores para los datos de entrada (por ejemplo. «el número estará comprendido entre 1 y 49»), se creará una clase válida ($1 \leq \text{número} \leq 49$) y dos clases no válidas ($\text{número} < 1$ y $\text{número} > 49$).
- R2.- Si se especifica un número de valores (por ejemplo. «se pueden registrar de uno a tres propietarios de un piso»), se creará una clase válida ($1 \leq \text{propietarios} \leq 3$) y dos no válidas ($\text{propietarios} < 1$ y $\text{propietarios} > 3$).
- R3.- Si se especifica una situación del tipo «debe ser» o booleana (por ejemplo, «el primer carácter debe ser una letra»), se identifican una clase válida («es una letra») y una no válida («no es una letra»).
- R4.- Si se especifica un conjunto de valores admitidos (por ejemplo, «pueden registrarse tres tipos de inmuebles: pisos, chalés y locales comerciales») Y se sabe que el programa trata de forma diferente cada uno de ellos, se identifica una clase válida por cada valor (en este caso son tres: piso, chalé y local) y una no válida (cualquier otro caso: por ejemplo, plaza de garaje).
- R5.- En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

La aplicación de estas reglas para la derivación de clases de equivalencia permite desarrollar los casos de prueba para cada elemento de datos del dominio de entrada. La división en clases deberían realizarla personas independientes al proceso de desarrollo del programa, ya que. si lo hace la persona que preparó la especificación o diseño el software, la existencia de algunas clases (en concreto, las no consideradas en el tratamiento) no será, probablemente, reconocida.

El último paso del método es el uso de las clases de equivalencia para identificar los casos de prueba correspondientes. Este proceso consta de las siguientes fases:

1. Asignación de un número único a cada clase de equivalencia.

2. Hasta que todas las clases de equivalencia hayan sido cubiertas por (incorporadas a) casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para cada una de las clases no válidas sin cubrir.

La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares. Por ejemplo, en un programa donde hay que «introducir cantidad (1-99) y letra inicial (A-Z)» ante el caso «105 &» (dos errores), se puede indicar sólo el mensaje «105 fuera de rango de cantidad» y dejar sin examinar el resto de la entrada (el error de introducir «&» en vez de una letra). En otro caso, más frecuente, el programa respondería con un ambiguo «dato incorrecto, introduzca valor» que podría fácilmente hacer creer al equipo de pruebas que los errores fueron correctamente identificados cuando sólo se analizó uno.

Veamos un ejemplo de aplicación de la técnica. Se trata de una aplicación bancaria en la que el operador deberá proporcionar un código, un nombre para que el usuario identifique la operación (por ejemplo, «nómina») y una orden que disparará una serie de funciones bancarias.

Especificación

1. Código área: número de 3 dígitos que no empieza por 0 ni por 1.
2. Nombre de identificación: 6 caracteres.
3. Órdenes posibles: «cheque», «depósito», «pago factura», «retirada de fondos».

Aplicación de las reglas

1. Código
 - número, regla 3, booleana:
 - clase válida (número)
 - clase no válida (no es número)
 - regla 5. la clase número debe subdividirse; por la regla 1, rango, obtenemos:
 - subclase válida (200 código 999)
 - dos subclases no válidas (código < 200; código > 999)
2. Nombre de id.. número específico de valores, regla 2:
 - clase válida (6 caracteres)
 - dos clases no válidas (más de 6; menos de 6 caracteres)
3. Orden, conjunto de valores, regla 4:
 - una clase válida para cada orden (“cheque”, “depósito”...); 4 en total.
 - una clase no válida “divisas”, por ejemplo).

En la tabla siguiente se han enumerado las clases identificadas y la generación de casos (presuponiendo que el orden de entrada es código-nombre-orden) se ofrece a continuación.

Condición de entrada	Clases válidas	Clases inválidas
Código área	(1) 200_código_999	(2) código <200

		(3) código >999 (4) no es número
Nombre para identificar la operación	(5) seis caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) «cheque» (9) «depósito» (10) « pago factura» (11) «retirada fondos»	(12) ninguna orden válida

Tabla de clases de equivalencia del ejemplo

Casos válidos:

- 300 Nómina Depósito (1) (5) (9)
- 400 Viajes Cheque (1) (5) (8)
- 500 Coches Pago-factura (1) (5) (10)
- 600 Comida Retirada-fondos (1) (5) (11)

Casos no válidos;

- 180 Viajes Pago-factura (2) (5) (10)
- 1032 Nómina Depósito (3) (5) (9)
- XY Compra Retirada-fondos (4) (5) (11)
- 350 A Depósito (1) (6) (9)
- 450 Regalos Cheque (1) (7) (8)
- 550 Casa &%4 (1) (5) (12)

6.6.2.- Análisis de Valores Límite (AVL)

Mediante a experiencia (e incluso a través de demostraciones) se ha podido constatar que los casos de prueba que exploran las condiciones límite de un programa producen un mejor resultado para la detección de defectos, es decir, es más probable que los defectos del software se acumulen en estas condiciones. Podemos definir las condiciones límite como las situaciones que se hallan directamente arriba, abajo y en los márgenes de las clases de equivalencia.

El análisis de valores límite es un técnica de diseño de casos que complementa a la de particiones de equivalencia. Las diferencias entre ambas son las siguientes:

- Más que elegir «cualquier» elemento como representativo de una clase de equivalencia. se requiere la selección de uno o más elementos tal que os márgenes se sometan a prueba.
- Más que concentrarse únicamente en el dominio de entrada (condiciones de entrada) los casos de prueba se generan considerando también el espacio de salida.

El proceso de selección de casos es también heurístico, aunque existen ciertas reglas orientativas. Aunque parezca que el AVL es simple de usar (a la vista de las reglas), su

aplicación tiene múltiples matices que requieren un gran cuidado a la hora de diseñar las pruebas. Las reglas para identificar clases son las siguientes:

1. Si una condición de entrada especifica un rango de valores («-1.0 valor +1.0») se deben generar casos para los extremos del rango (-1.0 y +1.0) y casos no válidos para situaciones justo más allá de los extremos (-1001 y +1.001, en el caso en que se admitan 3 decimales)
2. Si la condición de entrada especifica un número de valores («el fichero de entrada tendrá de 1 a 255 registros»), hay que escribir casos para los números máximo, mínimo, uno más del máximo y uno menos del mínimo de valores (0, 1, 255 y 256 registros).
3. Usar la regla 1 para la condición de salida «<el descuento máximo aplicable en compra al contado será el 50%, el mínimo será el 6%>». Se escribirán casos para intentar obtener descuentos de 5,99%, 6%, 50% y 50,01%.
4. Usar la regla 2 para cada condición de salida («el programa puede mostrar de 1 a 4 listados»). Se escriben casos para intentar generar 0, 1,4 y 5 listados.

En esta regla, como en la 3, debe recordarse que:

- Los valores límite de entrada no generan necesariamente los valores límite de salida (recuérdese la función seno, por ejemplo).
- No siempre se pueden generar resultados fuera del rango de salida (pero es interesante considerarlo).
- Si la entrada o la salida de un programa es un conjunto ordenado (por ejemplo, una tabla, un archivo secuencial, etc.). los casos se deben concentrar en el primero y en el último elemento.

Es recomendable utilizar el ingenio para considerar todos los aspectos y matices, a veces sutiles, en la aplicación del AVL.

6.6.3.- Conjetura de errores

La idea básica de esta técnica consiste en enumerar una lista de equivocaciones que pueden cometer los desarrolladores y de las situaciones propensas a ciertos errores. Despues se generan casos de prueba en base a dicha lista (se suelen corresponder con defectos que aparecen comúnmente y no con aspectos funcionales). Esta técnica también se ha denominado generación de casos (o valores) especiales, ya que no se obtienen en base a otros métodos sino mediante la intuición o la experiencia.

No existen directrices eficaces que puedan ayudar a generar este tipo de casos, ya que lo único que se puede hacer es presentar algunos ejemplos típicos que reflejan esta técnica. Algunos valores a tener en cuenta para los casos especiales son los siguientes:

1. El valor cero es una situación propensa a error tanto en la salida como en la entrada.
2. En situaciones en las que se introduce un número variable de valores (por ejemplo, una lista), conviene centrarse en el caso de no introducir ningún valor y en el de un solo valor. También puede ser interesante una lista que tiene todos los valores iguales.

3. Es recomendable imaginar que el programador pudiera haber interpretado algo mal en la especificación.
4. También interesa imaginar lo que el usuario puede introducir como entrada a un programa. Se dice que se debe prever toda clase de acciones de un usuario como si fuera «completamente tonto» o, incluso, como si quisiera sabotear el programa.

6.6 4.- Pruebas aleatorias

En las pruebas aleatorias simulamos la entrada habitual del programa creando datos de entrada en la secuencia y con la frecuencia con las que podrían aparecer en la práctica de forma continua: Esto implica usar una herramienta denominada generador de pruebas, a las que se alimenta con una descripción de las entradas y las secuencias de entrada posibles y su probabilidad de ocurrir en la práctica. Este enfoque de prueba es muy común en la prueba de compiladores en la que se generan aleatoriamente códigos de programas que sirven de casos de prueba para la compilación.

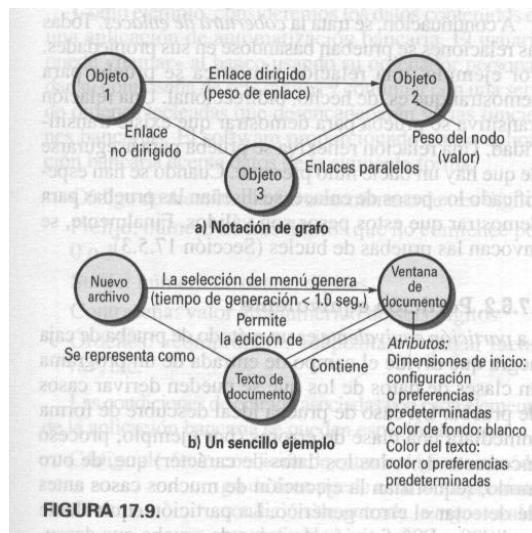
Si el proceso de generación se ha realizado correctamente, se crearán todas las posibles entradas del programa en todas las posibles combinaciones y, permutaciones. También se puede conseguir, indicando la distribución estadística que siguen, que la frecuencia de las entradas sea la apropiada para orientar correctamente nuestras pruebas hacia lo que es probable que suceda en la práctica. No obstante, esta forma de diseñar casos de prueba es menos utilizada que las de caja blanca y de caja negra

6.6.5.- Métodos de prueba basados en grafos

El primer paso en la prueba de caja negra es entender los objetos que se modelan en el software y las relaciones que conectan a estos objetos. Una vez que se ha llevado a cabo esto, el siguiente paso es definir una serie de pruebas que verifiquen que «todos los objetos tienen entre ellos las relaciones esperadas». Dicho de otra manera, la prueba del software empieza creando un grafo de objetos importantes y sus relaciones, y después diseñando una serie de pruebas que cubran el grafo de manera que se ejercent todos los objetos y sus relaciones para descubrir los errores.

Para llevar a cabo estos pasos, el ingeniero del software empieza creando un *grafo* —una colección de *nodos* que representan objetos: *enlaces* que representan las relaciones entre los objetos: *pesos de nodos* que describen las propiedades de un nodo (por ejemplo, un valor específico de datos o comportamiento de estado) y *pesos de enlaces* que describen alguna característica de un enlace—.

En la Figura 17.9a se muestra una representación simbólica de un grafo. Los nodos se representan como círculos conectados por enlaces que toman diferentes formas. Un *enlace dirigido* (representado por una flecha) indica que una relación se mueve sólo en una dirección.



Un *enlace bidireccional*, también denominado *enlace simétrico*, implica que la relación se aplica en ambos sentidos. Los enlaces paralelos se usan cuando se establecen diferentes relaciones entre los nodos del grafo.

Como ejemplo sencillo, consideremos una parte de un grafo de una aplicación de un procesador de texto (Fig. 17.9b) donde:

objeto n° 1 = selección en el menú de archivo nuevo

objeto n° 2 = ventana del documento

objeto n° 3 = texto del documento

Como se muestra en la figura, una selección del menú en archivo nuevo genera una ventana del documento. El peso del nodo de ventana del documento proporciona una lista de los atributos de la ventana que se esperan cuando se genera una ventana. El peso del enlace indica que la ventana se tiene que generar en menos de 1.0 segundos. Un enlace no dirigido establece una relación simétrica entre selección en el menú de archivo nuevo y texto del documento, y los enlaces paralelos indican las relaciones entre la ventana del documento y el texto del documento. En realidad, se debería generar un grafo bastante más detallado como precursor al diseño de casos de prueba. El ingeniero del software obtiene entonces casos de prueba atravesando el grafo y cubriendo cada una de las relaciones mostradas. Estos casos de prueba están diseñados para intentar encontrar errores en alguna de las relaciones.

Beizer describe un número de métodos de prueba de comportamiento que pueden hacer uso de los grafos:

Modelado del flujo de transacción. Los nodos representan los pasos de alguna transacción (por ejemplo, los pasos necesarios para una reserva en una línea aérea usando un servicio en línea), y los enlaces representan las conexiones lógicas entre los pasos (por ejemplo, *vuelo.información.entrada* es seguida de *validación/disponibilidad.procesamiento*). El diagrama de flujo de datos puede usarse para ayudar en la creación de grafos de este tipo.

Modelado de estado finito. Los nodos representan diferentes estados del software observables por el usuario (por ejemplo, cada una de las «pantallas» que aparecen

cuento un telefonista coge una petición por teléfono), y los enlaces representan las transiciones que ocurren para moverse de estado a estado (por ejemplo, *petición-información se verifica durante inventario-disponibilidad-búsqueda* y es seguido por *cliente-factura-información-entrada*). El diagrama estado-transición puede usarse para ayudar en la creación de grafos de este tipo.

Modelado del flujo de datos. Los nodos son objetos de datos y los enlaces son las transformaciones que ocurren para convertir un objeto de datos en otro. Por ejemplo, el nodo FICA.impuesto.retenido (FIR) se calcula de brutos.sueldos (BS) usando la relación FIR = 0,62 X BS.

Un estudio detallado de cada uno de estos métodos de prueba basados en grafos se sale de nuestros objetivos. Merece la pena, sin embargo, proporcionar un resumen genérico del enfoque de pruebas basadas en grafos.

Las pruebas basadas en grafos empiezan con la definición de todos los nodos y pesos de nodos. O sea, se identifican los objetos y los atributos. El modelo de datos puede usarse como punto de partida, pero es importante tener en cuenta que muchos nodos pueden ser objetos de programa (no representados explícitamente en el modelo de datos). Para proporcionar una indicación de los puntos de inicio y final del grafo, es útil definir unos nodos de entrada y salida.

Una vez que se han identificado los nodos se deberían establecer los enlaces y los pesos de enlaces. En general, conviene nombrar los enlaces, aunque los enlaces que representan el flujo de control entre los objetos de programa no es necesario nombrarlos.

En muchos casos, el modelo de grafo puede tener bucles (por ejemplo, un camino a través del grafo en el que se encuentran uno o más nodos más de una vez). La prueba de bucle se puede aplicar también a nivel de comportamiento (de caja negra). El grafo ayudará a identificar aquellos bucles que hay que probar.

Cada relación es estudiada separadamente, de manera que se puedan obtener casos de prueba. La *transitividad* de relaciones secuenciales es estudiada para determinar cómo se propaga el impacto de las relaciones a través de objetos definidos en el grafo. La transitividad puede ilustrarse considerando tres objetos X, Y y Z. Consideremos las siguientes relaciones:

- X es necesaria para calcular Y
- Y es necesaria para calcular Z

Por tanto, se ha establecido una relación transitiva entre X y Z:

- X es necesaria para calcular Z

Basándose en esta relación transitiva, las pruebas para encontrar errores en el cálculo de Z deben considerar una variedad de valores para X e Y.

La *simetría* de una relación (enlace de grafo) es también una importante directriz para diseñar casos de prueba. Si un enlace es bidireccional (simétrico), es importante probar esta característica. La característica *UNDO* (deshacer) en muchas aplicaciones

para ordenadores personales implementa una limitada simetría. Es decir, *UNDO* permite deshacer una acción después de haberse completado. Esto debería probarse minuciosamente y todas las excepciones (por ejemplo, lugares donde no se puede usar *UNDO*) deberían apuntarse. Finalmente, todos los nodos del grafo deberían tener una relación que los lleve devuelta a ellos mismos; en esencia un bucle de «no acción» o «acción nula». Estas relaciones *reflexivas* deberían probarse también.

Cuando empieza el diseño de casos de prueba, el primer objetivo es conseguir la *cobertura de nodo*. Esto significa que las pruebas deberían diseñarse para demostrar que ningún nodo se ha omitido inadvertidamente y que los pesos de nodos (atributos de objetos) son correctos.

A continuación, se trata la *cobertura de enlaces*. Todas las relaciones se prueban basándose en sus propiedades. Por ejemplo, una relación simétrica se prueba para demostrar que es, de hecho, bidireccional. Una relación transitiva se prueba para demostrar que existe transitividad. Una relación reflexiva se prueba para asegurarse de que hay un bucle nulo presente. Cuando se han especificado los pesos de enlace, se diseñan las pruebas para demostrar que estos pesos son válidos. Finalmente, se invocan las pruebas de bucles.

6.7.- Enfoque práctico recomendado para el diseño de casos

Los dos enfoques estudiados, caja blanca y caja negra, representan aproximaciones diferentes para las pruebas. El enfoque práctico recomendado para el uso de las técnicas de diseño de casos pretende mostrar el uso más apropiado de cada técnica para la obtención de un conjunto de casos útiles sin perjuicio de las estrategias de niveles de prueba

1. Si la especificación contiene combinaciones de condiciones de entrada, comenzar formando sus grafos de causa-efecto (ayudan a la comprensión de dichas combinaciones).
2. En todos los casos, usar el análisis de valores-límites para añadir casos de prueba: elegir límites para dar valores a las causas en Los casos generados asumiendo que cada causa es una clase de equivalencia.
3. Identificar las clases válidas y no válidas de equivalencia para la entrada y la salida, y añadir los casos no incluidos anteriormente (¿cada causa es una única clase de equivalencia? ¿Deben dividirse en clases?).
4. Utilizar la técnica de conjetura de errores para añadir nuevos casos, referidos a valores especiales.
5. Ejecutar los casos generados hasta el momento (de caja negra) y analizar la cobertura obtenida (en este punto ofrecen una gran ayuda las herramientas de análisis de cobertura).
6. Examinar la lógica del programa para añadir los casos precisos (de caja blanca) para cumplir el criterio de cobertura elegido si los resultados de La ejecución del punto 5 indican que no se ha satisfecho el criterio de cobertura elegido (que figura en el plan de pruebas).

Aunque éste es el enfoque integrado para una prueba «razonable», en la práctica la aplicación de las distintas técnicas está bastante discriminada según la etapa de la

estrategia de prueba.

Otra cuestión importante en relación a los dos tipos de diseño de pruebas es: ¿por qué incluir técnicas de caja blanca para explorar detalles lógicos y no centrarnos mejor en probar los requisitos funcionales con técnicas de caja negra? (si el programa realiza correctamente las funciones ¿por qué hay que intentar probar un determinado número de caminos?).

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa (a menor probabilidad de ejecutarse un camino, mayor número de errores).
- Se suele creer que un determinado camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar regularmente.
- Los errores tipográficos son aleatorios; pueden aparecer en cualquier parte del programa (sea muy usada o no).
- La probabilidad y La importancia de un trozo de código suele ser calculada de forma muy subjetiva.

Debemos recordar que tanto La prueba exhaustiva de caja blanca como de caja negra son impracticables. ¿Bastaría, no obstante, una prueba exhaustiva de caja blanca solamente? Véase el siguiente programa:

```
IF (x+y+z)/3 = x
THEN print («X, Y y Z son iguales»)
ELSE print («X, Y y Z no son iguales»)
```

En este programa, una prueba exhaustiva de caja blanca (que pase por todos los caminos) no asegura necesariamente la detección de los defectos de su diseño. véase, por ejemplo, cómo los dos casos siguientes no detectan ningún problema en el programa:

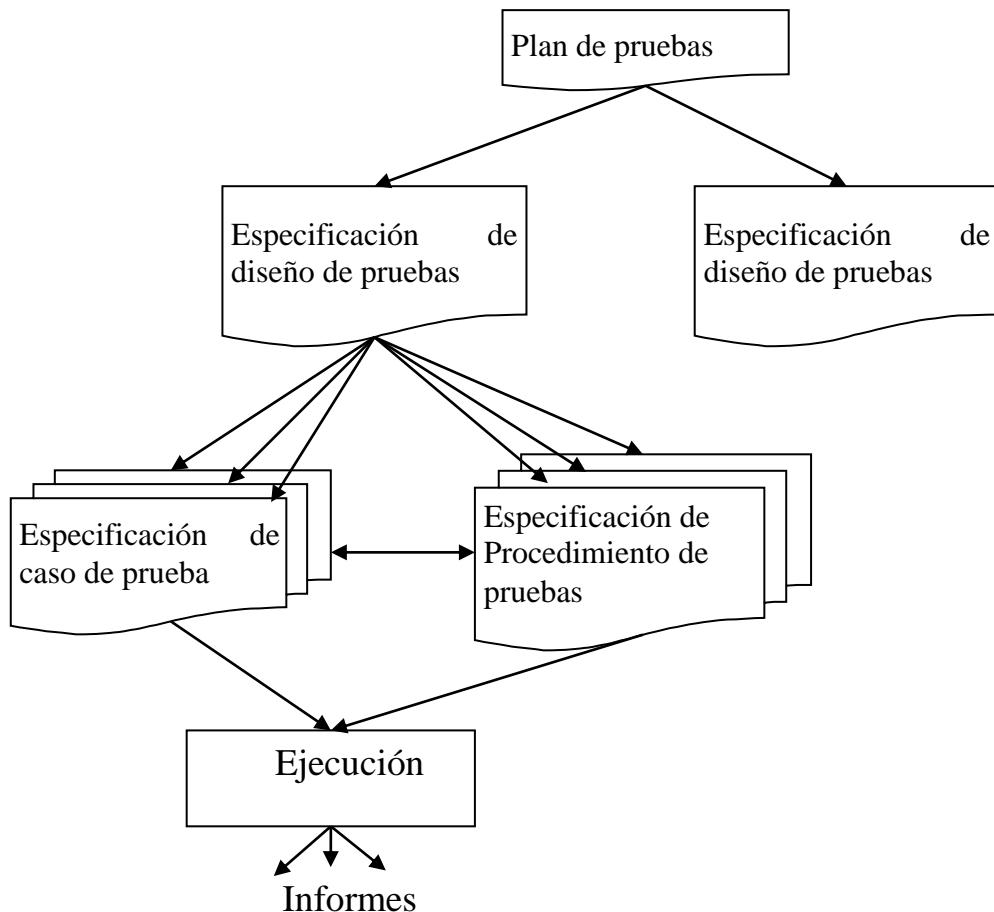
```
x=5, y=5, z=5
x=2, y=3, z=7
```

Sin embargo, el código tiene un problema. Véase el caso x=5, y=3, z=7, o x=4, y=5, z=3

Estos contraejemplos no pretenden influir en el tipo de técnica de diseño de casos que debemos elegir. Más bien nos indican que conviene emplear lo mejor de todas las técnicas para obtener pruebas más eficaces. Podemos citar una frase de B. Beizer a favor de las pruebas de caja blanca: «Los errores se esconden en los rincones y se acumulan en los límites».

6.8.- Documentación del diseño de las pruebas

Recordemos que la documentación de las pruebas es necesaria para una buena organización de las mismas, así como para asegurar su reutilización que es fundamental para optimizar tanto la eficacia como la eficiencia de las pruebas. Los distintos documentos de trabajo de las pruebas, según el estándar IEEE std. 829 son los que se reflejan en la figura.



Los documentos contemplados en el estándar se asocian a las distintas fases de las pruebas de la siguiente manera:

- El primer paso se sitúa en la planificación general del esfuerzo de prueba (plan de pruebas) para cada fase de la estrategia de prueba para el producto
- Se genera inicialmente la especificación del diseño de la prueba (que surge de la ampliación y el detalle del plan de pruebas).
- A partir de este diseño, se pueden definir con detalle cada uno de los casos mencionados escuetamente en el diseño de la prueba (se fijan los datos de prueba exactos, los resultados esperados, etc.).
- Tras generar los casos de prueba detallados, se debe especificar cómo proceder en detalle en la ejecución de dichos casos (procedimientos de prueba).
- Tanto las especificaciones de casos de prueba como las especificaciones de los procedimientos deben ser los documentos básicos para la ejecución de las pruebas. No obstante, son los procedimientos los que determinan realmente cómo se desarrolla la ejecución.

A continuación, describiremos brevemente los contenidos de los distintos documentos contemplados en el estándar.

6.8.1.- Plan de pruebas

Objetivo del documento: señalar el enfoque, los recursos y el esquema de actividades de prueba, así como los elementos a probar, las características, las actividades de prueba, el personal responsable y los riesgos asociados.

- *Estructura fijada en el estándar:*

1. Identificador único del documento (para la gestión de configuración).
2. Introducción y resumen de elementos y características a probar.
3. Elementos software que se van a probar (por ejemplo, programas o módulos).
4. Características que se van a probar.
5. Características que no se prueban.
6. Enfoque general de la prueba (actividades, técnicas, herramientas, etc.).
7. Criterios de paso/fallo para cada elemento.
8. Criterios de suspensión y requisitos de reanudación.
9. Documentos a entregar (como mínimo, los descritos en el estándar).
10. Actividades de preparación y ejecución de pruebas.
11. Necesidades de entorno.
12. Responsabilidades en la organización y realización de las pruebas.
13. Necesidades de personal y de formación.
14. Esquema de tiempos (con tiempos estimados, hitos, etc.).
15. Riesgos asumidos por el plan y planes de contingencias para cada riesgo.
16. Aprobaciones y firmas con nombre y puesto desempeñado.

6.8.2.- Especificación del diseño de pruebas

Objetivo del documento: especificar los refinamientos necesarios sobre el enfoque general reflejado en el plan e identificar las características que se deben probar con este diseño de pruebas.

Estructura fijada en el estándar:

1. Identificador (único) para la especificación Proporcionar también una referencia del plan asociado (si existe).
2. Características a probar de los elementos software (y combinaciones de características).
3. Detalles sobre el plan de pruebas del que surge este diseño, incluyendo las técnicas de prueba específica y los métodos de análisis de resultados.
4. Identificación de cada prueba:
 - Identificador.
 - Casos que se van a utilizar.
 - Procedimientos que se van a seguir
5. Criterios de paso/fallo de la prueba (criterios para determinar si una característica o combinación de características ha pasado con éxito la prueba o no).

6.8.3.- Especificación de caso de prueba

Objetivo del documento: definir uno de los casos de prueba identificado por una especificación del diseño de las pruebas.

Estructura fijada en el estándar:

1. Identificador único de la especificación.
2. Elementos software (por ejemplo, módulos que se van a probar: definir dichos elementos y las características que ejercitará este caso)
3. Especificaciones de cada entrada requerida para ejecutar el caso (incluyendo las relaciones entre las diversas entradas; por ejemplo, la sincronización de las mismas).
4. Especificaciones de todas las salidas y las características requeridas (por ejemplo, el tiempo de respuesta) para los elementos que se van a probar.
5. Necesidades de entorno (hardware, software y otras como, por ejemplo, el personal).
6. Requisitos especiales de procedimiento (o restricciones especiales en los procedimientos para ejecutar este caso).
7. Dependencias entre casos (por ejemplo, listar los identificadores de los casos que se van a ejecutar antes de este caso de prueba).

6.8.4.- Especificación de procedimiento de prueba

Objetivo del documento: especificar los pasos para la ejecución de un conjunto de casos de prueba o, más generalmente, los pasos utilizados para analizar un elemento software con el propósito de evaluar un conjunto de características del mismo.

Estructura fijada en el estándar:

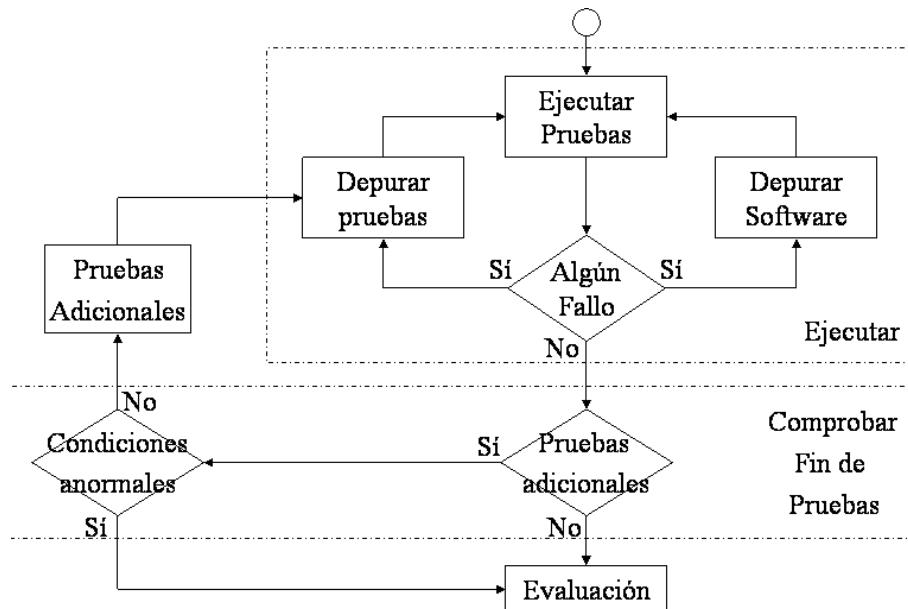
1. Identificador único de la especificación y referencia a la correspondiente especificación de diseño de prueba.
2. Objetivo del procedimiento y lista de casos que se ejecutan con él.
3. Requisitos especiales para la ejecución (por ejemplo, entorno especial o personal especial).
4. Pasos en el procedimiento. Además de la manera de registrar los resultados y los incidentes de la ejecución, se debe especificar:
 - La secuencia necesaria de acciones para preparar la ejecución
 - Acciones necesarias para empezar la ejecución.
 - Acciones necesarias durante la ejecución.
 - Cómo se realizarán las medidas (por ejemplo, el tiempo de respuesta>.
 - Acciones necesarias para suspender la prueba (cuando los acontecimientos no previstos lo obliguen).
 - Puntos para reinicio de la ejecución y acciones necesarias para el reinicio en estos puntos.
 - Acciones necesarias para detener ordenadamente la ejecución.

- Acciones necesarias para restaurar el entorno y dejarlo en la situación existente antes de las pruebas.
- Acciones necesarias para tratar los acontecimientos anómalos

6.9.- Ejecución de las pruebas

6.9.1.- El proceso de ejecución

El proceso de las pruebas según el estándar IEEE std. 1008 en el que se incluye la ejecución de pruebas es el mostrado en la figura.



El proceso abarca las siguientes fases:

- Ejecutar las pruebas, cuyos casos y procedimientos han sido ya diseñados previamente.
- Comprobar si se ha concluido el proceso de prueba (según ciertos criterios de compleción de prueba que suelen especificarse en el plan de pruebas)
- En el caso de que hayan terminado las pruebas, se evalúan los resultados; en caso contrario, hay que generar las pruebas adicionales para que se satisfagan los criterios de compleción de pruebas.

El proceso de ejecución propiamente dicho consta de las siguientes fases:

- Se ejecutan las pruebas, es decir, se realiza el paso por el ordenador de los datos de prueba
- Se comprueba si ha habido algún fallo al ejecutar (por ejemplo, se cae el sistema, se bloquea el teclado, etc., es decir, cualquier hecho que impide terminar la ejecución de algún caso).
- Si lo ha habido, se puede deber a un defecto software, lo que nos lleva al proceso de depuración o corrección del código. Se puede deber también a un

defecto en el propio diseño de las pruebas (por ejemplo, resulta imposible al sistema operativo, no al programa que se prueba, adaptarse a las condiciones o entorno que exigen las pruebas). En ambos casos, las nuevas pruebas o las corregidas se deberán ejecutar en el ordenador.

- De no existir fallo, se pasará a la comprobación de la terminación de las pruebas

En la figura se muestra el proceso de comprobación de la terminación de las pruebas. Se pueden observar las siguientes fases:

- Tras la ejecución, se comprobará si se cumplen los criterios de compleción de pruebas descritos en el correspondiente plan de pruebas (por ejemplo, se terminan las pruebas cuando se han probado todos los procedimientos de operación o se ha cumplido la cobertura lógica marcada).
- En caso de terminar las pruebas, se pasa a la evaluación de los productos probados sobre la base de los resultados obtenidos (terminación normal).
- En caso de no terminar las pruebas, se debe comprobar la presencia de condiciones anormales en la prueba (por ejemplo, un defecto importante no corregido ya detectado previamente, tiempo finalizado, etc.).
- Si hubiesen existido condiciones anormales (por ejemplo, no se han podido ejecutar todos los casos porque el sistema se cae regularmente), se pasa de nuevo a la evaluación (terminación anormal); en caso contrario se pasa a generar y ejecutar pruebas adicionales para satisfacer cualquiera de las dos terminaciones.

Los criterios de compleción de pruebas hacen referencia a las áreas (características, funciones, instrucciones, etc.) que deben cubrir las pruebas y el grado de cobertura para cada una de esas áreas. Como ejemplos genéricos de este tipo de criterios se pueden indicar los siguientes:

- Se debe cubrir cada característica del software mediante un caso de prueba o una excepción aprobada en el plan de pruebas.
- En programas codificados con lenguajes procedimentales, se deben cubrir todas las instrucciones ejecutables mediante casos de prueba (o se deben marcar as posibles excepciones aprobadas en el plan de pruebas).

6.9.2- Documentación de la ejecución de pruebas

Al igual que en el diseño de las pruebas, la documentación de la ejecución de las pruebas es fundamental para la eficacia en la detección y corrección de defectos, así como para dejar constancia de los resultados de las pruebas.

La figura siguiente nos muestra el conjunto de documentos que se relaciona directamente con la ejecución de las pruebas según el estándar IEEE std. 829. Se pueden distinguir dos grupos principales:

1. Documentación de entrada, constituida principalmente por las especificaciones de los casos de prueba que se van a usar y las especificaciones de los procedimientos de pruebas (ambas ya explicadas con anterioridad).

2. Documentación de salida o informes sobre la ejecución. Cada ejecución de pruebas generará dos tipos de documentos
 - Histórico de pruebas o registro cronológico de la ejecución.
 - Informes de los incidentes ocurridos (si hay) durante la ejecución.
3. La documentación de salida correspondiente a un mismo diseño de prueba se recoge en un informe resumen de pruebas.

6.9.3.- Histórico de pruebas

Objetivo: el histórico de pruebas (*test log*) documenta todos los hechos relevantes ocurridos durante la ejecución de las pruebas.

Estructura fijada en el estándar:

1. Identificador.
2. Descripción de la prueba: elementos probados y entorno de la prueba
3. Anotaciones de datos sobre cada hecho ocurrido (incluido el comienzo y el final de la prueba):
 - Fecha y hora.
 - Identificador de informe de incidente.
4. Otras informaciones

6.9.4.- Informe de incidente

Objetivo: el informe de incidente (*test incident report*) documenta cada incidente (por ejemplo, una interrupción en las pruebas debido a un corte de electricidad, bloqueo del teclado, etc.) ocurrido en la prueba y que requiera una posterior investigación.

Estructura fijada en el estándar:

1. Identificador.
2. Resumen del incidente.
3. Descripción de datos objetivos (fecha/hora, entradas, resultados esperados, etc.).
4. Impacto que tendrá sobre las pruebas.

6.9.5.- Informe resumen de las pruebas

Objetivo: el informe resumen (*test summary report*) resume los resultados de las actividades de prueba (las reseñadas en el propio informe) y aporta una evaluación del software basada en dichos resultados.

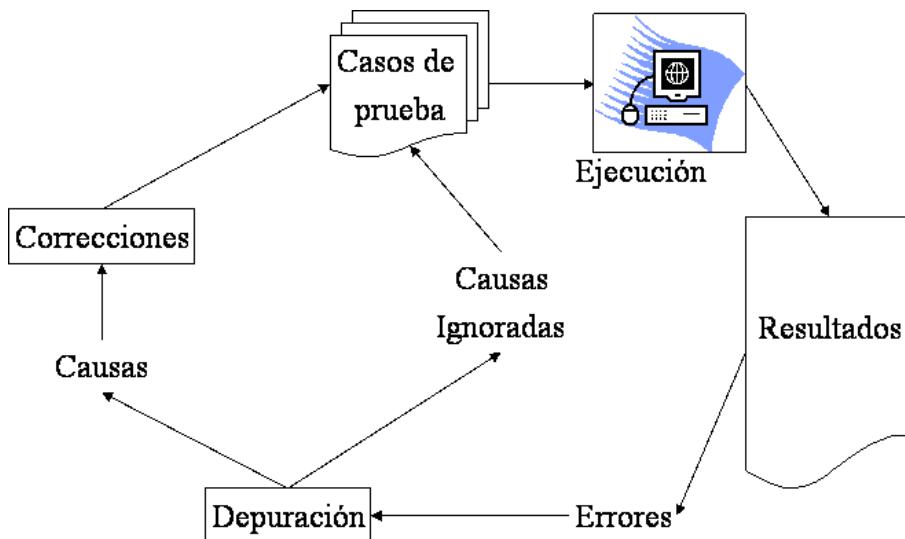
Estructura fijada en el estándar:

1. Identificador.
2. Resumen de la evaluación de los elementos probados.
3. Variaciones del software respecto a su especificación de diseño, así como las variaciones en las pruebas.
4. Valoración de la extensión de la prueba (cobertura lógica, funcional, de requisitos, etc.).
5. Resumen de los resultados obtenidos en las pruebas.

6. Evaluación de cada elemento software sometido a prueba (evaluación general del software incluyendo las limitaciones del mismo).
7. Resumen de las actividades de prueba (incluyendo el consumo de todo tipo de recursos).
8. Firmas y aprobaciones de quienes deban supervisar el informe.

6.9.6.- Depuración

Se define la depuración como «el proceso de localizar, analizar y corregir los defectos que se sospecha que contiene el software». Suele ser la consecuencia de una prueba con éxito (es decir, que descubre los síntomas de un defecto).



Las consecuencias de la depuración pueden ser dos (véase la figura):

- Encontrar la causa del error, analizarla y corregirla.
- No encontrar la causa y, por lo tanto, tener que generar nuevos casos de prueba que puedan proporcionar información adicional para su localización (casos de prueba para depuración).

Las dos principales etapas en la depuración son las siguientes:

- Localización del defecto, que conlleva la mayor parte del esfuerzo.
- Corrección del defecto, efectuando las modificaciones necesarias en el software.

El proceso de prueba (véase figura 12.13) implica generar unos casos de prueba, ejecutarlos en el ordenador y obtener unos resultados. Dichos resultados se analizan para la búsqueda de síntomas de defectos (errores) en el software. Esta información se pasa al proceso de depuración para obtener las causas del error (defecto). En caso de conseguirlo, se corrige el defecto; en caso contrario, se llevarán a cabo nuevas pruebas que ayuden a localizarlo (reduciendo en cada pasada el posible dominio de existencia del defecto). Tras corregir el defecto, se efectuarán nuevas pruebas que comprueben si se ha eliminado dicho problema.

6.9.6.1.- Consejos para la depuración

G. J. Myers [MYERS. 1979] aportó la siguiente lista de consejos para el proceso de depuración:

- *Localización del error*

1. **Analizar la información y pensar.** La depuración es un proceso mental de resolución de un problema y lo mejor para el mismo es el análisis. No se debe utilizar un enfoque aleatorio en la búsqueda del defecto.
2. **Al llegar a un punto muerto, pasar a otra cosa.** Si tras un tiempo razonable no se consiguen resultados, merece la pena refrescar la mente (para empezar de nuevo o para que el inconsciente nos proporcione la solución).
3. **Al llegar a un punto muerto, describir el problema a otra persona.** El simple hecho de describir a otro el problema nos descubre muchas cosas («cuando todo falle, pedir ayuda»).
4. **Usar herramientas de depuración sólo como recurso secundario.** Deben ayudar al análisis mental, no pueden pretender sustituirlo (por lo menos, en la actualidad).
5. **No experimentar cambiando el programa.** Evitar depurar con esta actitud inadecuada que se puede resumir como: «No sé qué está mal, así que cambiaré este bucle y veré qué pasa».
6. **Se deben atacar los errores individualmente,** de uno en uno, so pena de dificultar aún más la depuración
7. **Se debe fijar la atención también en los datos** manejados en el programa y no sólo en la lógica del proceso.

- *Corrección del error*

1. **Donde hay un defecto, suele haber más.** (Principio de Pareto) Es una conclusión obtenida de la experiencia. Cuando se corrige un defecto se debe examinar su proximidad inmediata buscando elementos sospechosos.
2. **Debe corregirse el defecto, no sus síntomas.** Lo que debe corregirse y desaparecer es el defecto, no se trata de intentar enmascarar sus síntomas.
3. **La probabilidad de corregir perfectamente un defecto no es del 100%.** Por lo tanto, se deben revisar las correcciones antes de implantarlas (mediante técnicas de revisión: walkthroughs, inspecciones, revisiones, etc., antes de la corrección). Después de la corrección, se utilizan pruebas específicas.
4. **Cuidado con crear nuevos defectos.** Es frecuente crear nuevos defectos al corregir sin cautela. La probabilidad de que un cambio se realice correctamente es del 50% (aproximadamente) según algunos estudios.
5. **La corrección debe situarnos temporalmente en la fase de diseño.** Hay que mentalizarse de que se reinicia el diseño de la sección de código defectuoso y no sólo se retoca el código.
6. **Cambiar el código fuente, no el código objeto.** Cambiar el código objeto provoca:
 - experimentación indeseable.
 - falta de sincronización fuente-objeto.

6.9.7.- Análisis de errores a análisis causal

El objetivo del análisis causal es proporcionar información sobre la naturaleza de los defectos (de los cuales sabemos muy poco). Para ello, es fundamental que se recoja la siguiente información de cada defecto detectado:

1. ¿Cuándo se cometió?
2. ¿Quién lo hizo?
3. ¿Qué se hizo mal?
4. ¿Cómo se podría haber prevenido?
5. ¿Por qué no se detectó antes?
6. ¿Cómo se podría haber detectado antes?
7. ¿Cómo se encontró el error?

El objetivo primordial es la formación del personal sobre los errores que comete para que se puedan prevenir en futuro. Nunca debe usarse esta información para evaluar al personal o para «buscar a quién despedir». El mantenimiento y almacenamiento de un archivo de los defectos detectados y corregidos, anotando dónde ocurrieron los errores y los tipos de defectos encontrados, se puede utilizar para la predicción de los futuros fallos del software. Esta predicción se basa en complejos modelos de fiabilidad que requieren unos conocimientos matemáticos considerables.

6.10.- Estrategia de aplicación de las pruebas

Una vez conocidas las técnicas de diseño y ejecución, debemos analizar cómo se plantea la utilización de las pruebas en el ciclo de vida. La estrategia de aplicación y la planificación de las pruebas pretenden integrar el diseño de los casos de prueba en una serie de pasos bien coordinados a través de la creación de distintos niveles de prueba con diferentes objetivos. Además, permite la coordinación del personal de desarrollo, del departamento de aseguramiento de calidad (o algún grupo independiente dedicado a la V y V o a las pruebas) y del cliente, gracias a la definición de los papeles que deben desempeñar cada uno de ellos y la forma de llevarlos a cabo.

En general, la estrategia de pruebas suele seguir las siguientes etapas:

- Las pruebas comienzan a nivel de módulo.
- Una vez terminadas, progresan hacia la integración del sistema completo y su instalación.
- Culminan cuando el cliente acepta el producto y se pasa a su explotación inmediata.

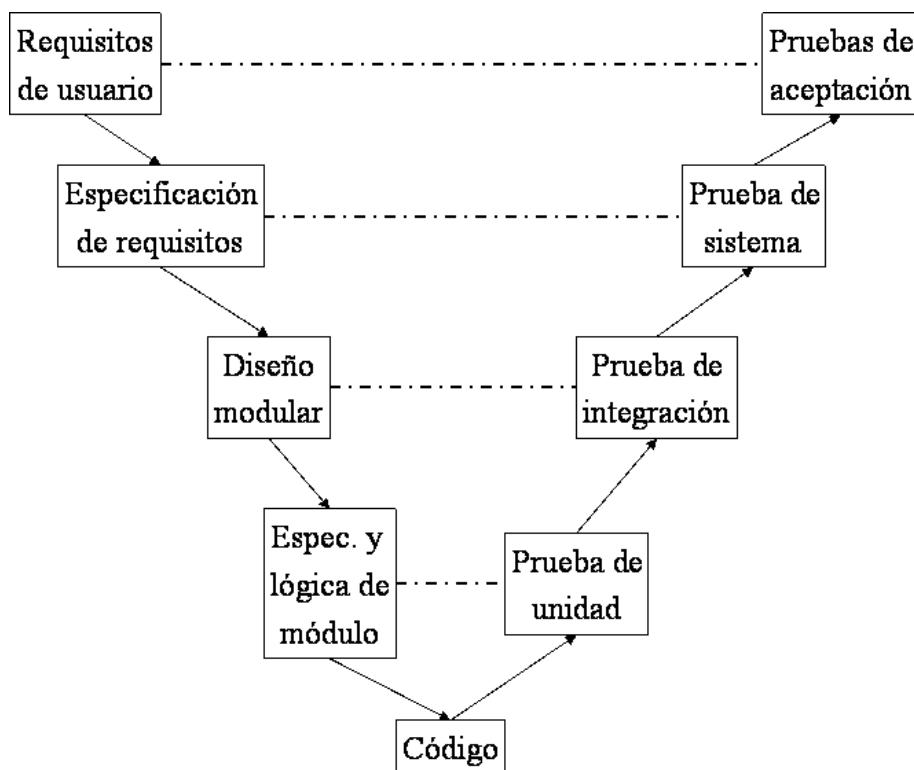
Esta serie típica de etapas se describe con mayor detalle a continuación:

1. Se comienza en la prueba de cada módulo, que normalmente la realiza el propio personal de desarrollo en su entorno.
2. Con el esquema del diseño del software, los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto (prueba de integración).
3. El software totalmente ensamblado se prueba como un conjunto para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimientos, seguridad, etc. (prueba funcional o de validación). Este nivel

de prueba coincide con el de la prueba del sistema cuando no se trate de software empotrado u otros tipos de especiales de aplicaciones.

4. El software ya validado se integra con el resto del sistema (por ejemplo, elementos mecánicos interfaces electrónicas, etc.) para probar su funcionamiento conjunto (prueba del sistema).
5. Por último, el producto final se pasa a la prueba de aceptación para que el usuario compruebe en su propio entorno de explotación si lo acepta como está o no (prueba de aceptación)

Lo cierto es que cada nivel de prueba se centra en probar el software en referencia al trabajo realizado en una diferente etapa de desarrollo. Así (véase la figura) dicha relación se concreta en un modelo de ciclo de vida denominado ciclo en uve.



1. La prueba de módulo (prueba de unidad) centra sus actividades en ejercitarse la lógica del módulo (caja blanca) y los distintos aspectos de la especificación de las funciones que debe realizar el módulo (caja negra).
2. La prueba de interacción debe tener en cuenta los mecanismos de agrupación de módulos fijados en la estructura del programa, así como, en general, las interfaces entre componentes de la arquitectura del software.
3. La prueba de validación (o funcional) debe comprobar si existen desajustes entre el software y los requisitos fijados para su funcionamiento en la ERS.
4. La prueba del sistema debe centrar sus comprobaciones en el cumplimiento de los objetivos indicados para el sistema.
5. La prueba de aceptación sirve para que el usuario pueda verificar si el producto final se ajusta a los requisitos fijados por él (normalmente en forma

de criterios de aceptación en el contrato) o, en último caso, en función de lo que indique el contrato.

6.11.- Pruebas en desarrollos orientados a objetos

Las técnicas y estrategias presentadas en este capítulo están inspiradas en la aplicación a desarrollos estructurados. Cuando trabajamos con tecnología orientada a objetos, algunas de las técnicas presentadas pierden su interés o deben adaptarse para seguir resultando útiles. Desde el punto de vista del diseño de casos de pruebas:

- Las técnicas de caja negra tienen vigencia toda vez que su uso se centra en el comportamiento de la aplicación. En el caso de pruebas de sistema, son las descripciones de casos de uso las que pueden aportar la referencia sobre la cual diseñar los casos de prueba. De hecho, los escenarios de casos de Uso permiten definir los casos de prueba aplicando técnicas de caja negra sobre los datos y los eventos incluidos en los escenarios. Éstos, a su vez, se pueden identificar en función de los caminos que se pueden trazar en los diagramas de actividad o en los diagramas de estados descriptivos del caso, sin olvidar todos los flujos alternativos o tratamientos de error y excepciones. Evidentemente, la aplicación de valores límite, tratamiento de combinaciones de entrada y conjectura de errores es perfectamente aplicable al software orientado a objetos.
- Las técnicas de caja blanca disminuyen sus posibilidades de aplicación, ya que quedan confinadas a las instrucciones de cada método de cada clase para conseguir el nivel de cobertura más apropiado. Evidentemente la cobertura de sentencias resulta esencial y el tratamiento de bucles y decisiones a través de las coberturas de condiciones y decisiones es más que recomendable. También cabría la posibilidad de aplicar la cobertura de caminos (incluido el criterio de McCabe) no sólo al código de método sino a diagramas de comportamiento con estructura de red: diagramas de estados, de actividad, etc.

En el caso del diseño de pruebas cabe señalar la posibilidad de distinguir criterios según el nivel de pruebas en el que nos situemos. Así, para las pruebas de unidad, es decir, pruebas de clases, la prueba de clases de equivalencia y valores límite es apropiada aplicándola a las entradas y salidas representadas como parámetros de métodos fundamentalmente. Un enfoque basado en clasificar operaciones según el diagrama de estados de la clase también resulta válido.

Sin embargo, los cambios más radicales aparecen en las pruebas de integración, cuando nos fijamos en la interacción entre objetos de distintas clases. El esquema de integración basado en la existencia de una estructura jerárquica que permita unir elementos de forma ascendente o descendente no tiene sentido en las estructuras de colaboración en red definidas para el software orientado a objetos. Para realizar la integración existen propuestas que permiten encontrar aquellas clases que dependen en su funcionamiento de sí mismas o sólo de unas pocas clases para tomarlas como destino de la primera oleada de pruebas de integración (después se irían sometiendo a prueba las clases que dependen sólo de las ya probadas en la primera tanda y así consecutivamente hasta integrar toda la aplicación).

Sin embargo, el enfoque más habitual consiste en ir probando hilos de clases que colaboran para una función o servicio del sistema (seguramente definidos en los diagramas de colaboración). Siempre habrá que asegurar todos los conjuntos de colaboración con todos los métodos de todas las clases de la aplicación. De nuevo, los diagramas de estados pueden apoyar una mayor información para el rigor de la prueba.

Para terminar, señalaremos dos cuestiones que tienen una influencia capital en las pruebas de software orientado a objetos:

- Uno de los problemas que provoca la tecnología de objetos en las pruebas es el tratamiento de la herencia. Las dificultades proceden del hecho de que la prueba de un método, que es heredado en clases inferiores, suponga probar no sólo el método genérico sino también todas las implementaciones en las clases hijas y en todo el árbol de herencia que genera. Además, el polimorfismo interviene para complicar las pruebas. Hay, entonces, que generar casos de prueba adicionales para cada implementación además de los que se heredan o solapan con los de la prueba en las clases padre.
- También es conveniente recordar que la propia programación o diseño orientado a objetos (con conceptos como encapsulación, herencia, etc.) hace, en relación con los desarrollos estructurados, menos probables ciertos tipos de error (por lo que los casos de prueba dedicados a detectarlos deberán disminuir), más probables otros (que deberán ser más vigilados en las pruebas) y la aparición de nuevos tipos de defectos.