	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.2 VIGENCIA: 19-09-2007
APUNTE DE TEMAS DE REVISIÓN		

PROCESO DE SOFTWARE

Proceso de resolución de problemas

- (1) Identificar el problema
- (2) Definir y representar el problema
- (3) Explorar las posibles estrategias
- (4) Aplicar y mejorar las estrategias
- (5) Mirar atrás y evaluar los efectos de la actividad realizada (¿se ha resuelto el problema?)

Este proceso es semejante a la siguiente secuencia, más simple:

- (1) Decidir **qué** hacer (cuál es el problema)
- (2) Decidir **cómo** hacerlo
- (3) Hacerlo
- (4) **Probar** el resultado
- (5) **Usar** el resultado

Por supuesto, la aplicación de este proceso general no resuelve los problemas. Precisamente por estar situado a un nivel demasiado abstracto. La mayor parte de las dificultades a la hora de resolver problemas surgen en los niveles más concretos, y no en los niveles abstractos. O lo que es lo mismo, la secuencia anterior describe el proceso seguido por la mente, y esto ciertamente ayuda, pero son necesarios muchos conocimientos del dominio, o área concreta donde se plantea el problema, para poder aplicar este proceso de modo que lleve a la solución.

El problema de la construcción de software

El proceso de construir un producto software es también una **actividad de resolución de problemas**. En realidad, la construcción de software tiene dos objetivos. Por un lado, dada una necesidad, pretende satisfacerla mediante una solución tratable por computadora, **Figura 1**. Por otro, el subsecuente mantenimiento del software producido hasta el final de su vida útil.

La solución de un problema mediante Ingeniería de Software es una actividad de modelización que comienza con el desarrollo de **modelos conceptuales** (no formales) y los convierte en **modelos formales**, que son los productos implementados, **Figura 2**. Estas dos actividades de modelización trabajan a niveles distintos: el **nivel del problema** o necesidad (nivel conceptual o de dominio de aplicación), y el **nivel de la solución** implementada sobre computadora (o nivel formal). En análisis de sistemas software, el modelo conceptual se corresponde con el punto de vista que las personas tienen del problema, en tanto que el modelo formal concierne a la perspectiva que de ese mismo problema tiene la computadora.



Figura 1

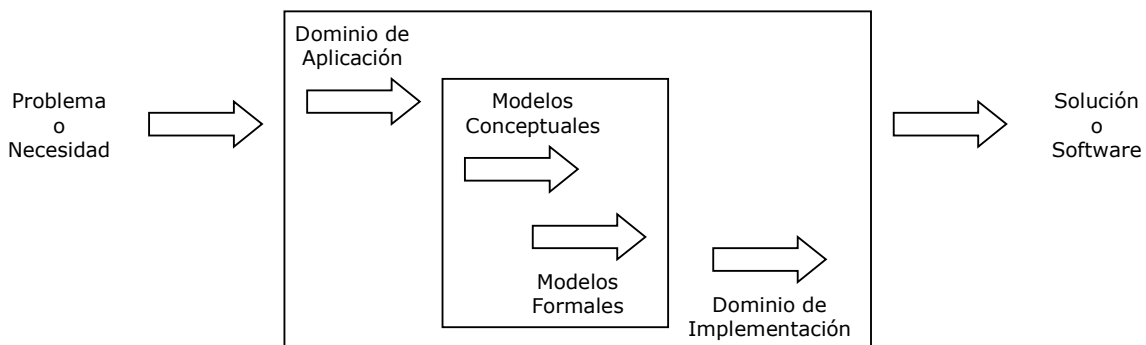


Figura 2

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

La **Figura 2** ilustra varios aspectos importantes del proceso de construcción de software. Lo fundamental que puede apreciarse en la **Figura 2**, es que los modelos conceptuales y formales son distintos y no puede derivarse lógicamente el modelo formal a partir del modelo conceptual. El modelo conceptual determina la validez (¿es válido el modelo obtenido para la necesidad que se tenía?), en tanto que el formal determina la corrección (¿funciona correctamente el modelo?). De hecho, es imposible establecer la corrección de un modelo conceptual pero, usando modelos formales del dominio, es posible descubrir errores que invaliden el modelo conceptual. En el dominio de la aplicación los modelos son conceptuales en el sentido de que modelizan cómo el software debería responder a una necesidad. Aunque pueden emplearse formalismos para representar estos modelos, desde la perspectiva de la computadora son, hasta el momento, modelos meramente declarativos o descriptivos, no operativos. La actividad de modelización formal comienza sólo después de que existe un modelo conceptual, o sea un modelo descriptivo de cómo el software responde a la necesidad. El modelo formal generado es un modelo prescriptivo o procedimental y computable por la máquina.

Además, la **Figura 2** muestra que el proceso software siempre culmina con la creación de un modelo formal. Pero, aunque la actividad de modelización formal es esencial y los métodos formales que preservan la corrección son importantes, esto no asegura el éxito de un proyecto. Cuando los sistemas son abiertos, las necesidades cambian y entonces la validez del modelo conceptual es un factor de riesgo. En efecto, la corrección del producto con respecto al modelo formal es, naturalmente condición necesaria pero difícilmente suficiente.

Por último, el modelo conceptual muestra por qué la construcción de software puede ser tan difícil. Existen muchas respuestas potenciales a una necesidad, y el modelo conceptual limita la elección a una. Raramente, hay una mejor elección; habitualmente, sólo hay malas elecciones a evitar. Además, para cada clase de modelo conceptual, se definen muchos modelos formales que pueden producir una respuesta satisfactoria, y muchas posibles programaciones correctas por cada modelo formal. De este modo, hay que trabajar en dos dominios, con dos tipos de modelos y gestionando elecciones que, necesariamente, están basadas en el juicio y la experiencia.

Planteadas la construcción de software como una resolución de problemas, debe existir un proceso de resolución que se corresponda con el proceso básico de resolución de problemas expuesto en el apartado anterior. En efecto, al primer paso, o definición del *qué*, se le denomina **Análisis y Especificación de Requisitos**. A la decisión de *cómo* hacerlo se la conoce, en Ingeniería de Software, como **Diseño del Sistema Software**. A la realización de ese *cómo* se le llama **Codificación**.

Posteriormente, el sistema debe ser sometido a **Pruebas**. Y, finalmente, la solución debe ser usada, o, en el caso del software, **Instalado**.

Además, cuando las soluciones a los problemas no son puntuales, sino que permanecen en el tiempo, al proceso de resolución debe añadirse una última etapa de **Mantenimiento**.

Por tanto, el **proceso** mínimo necesario para resolver el problema de la construcción de un sistema software es:

Obtención de requisitos software.

Incluye el análisis del problema y concluye con una especificación completa del comportamiento externo que debería tener el sistema a construir.

Diseñar.

El diseño del sistema debe realizarse a dos niveles: alto nivel o diseño preliminar, y bajo nivel o diseño detallado. En el diseño preliminar se descompone el sistema software en sus componentes principales, estos componentes se subdividen a su vez en componentes más pequeños. Este proceso iterativo continúa hasta un nivel adecuado en el que los componentes puedan ser tratados en el diseño de bajo nivel. Generalmente, estos módulos realizan una única función bien detallada y pueden venir descritos por su entrada, su salida y la función que realizan. En el diseño detallado se definen y documentan los algoritmos que llevarán a cabo la función a realizar por cada módulo.


Implementar.

Consiste en transformar los algoritmos definidos durante el diseño de bajo nivel en un lenguaje comprensible para una computadora. La codificación suele llevarse a cabo en dos niveles: la conversión del algoritmo en un lenguaje de alto nivel; y la transformación del lenguaje de alto nivel en lenguaje máquina. Generalmente, el primer nivel es realizado por personas y el segundo nivel se realiza automáticamente por un compilador.

Realizar pruebas.

Si los humanos fueran perfectos en el desarrollo del software, el proceso podría finalizar en el punto anterior. Desafortunadamente, éste no es el caso. Por lo tanto, se necesita un proceso de comprobación o pruebas para eliminar los errores. La comprobación se divide en tres niveles.

Pruebas unitarias, comprueban cada módulo implementado en busca de errores. En esta comprobación se quiere asegurar que cada módulo se comporta de acuerdo con lo especificado durante el diseño de bajo nivel. Pruebas de integración, que interconectan conjuntos de módulos previamente probados, asegurándose de que el conjunto se comporta tan bien como lo hacían independientemente. Lo ideal es que cada conjunto de módulos integrados se correspondiera con un componente del diseño de alto nivel. Pruebas del sistema, pretenden asegurar que la totalidad del sistema software (totalmente integrado en su entorno), se comporte de acuerdo con la especificación de requisitos inicial.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

Instalar.

Tras las pruebas, el sistema software y su entorno hardware pasan a la fase operativa.

Mantener y ampliar.

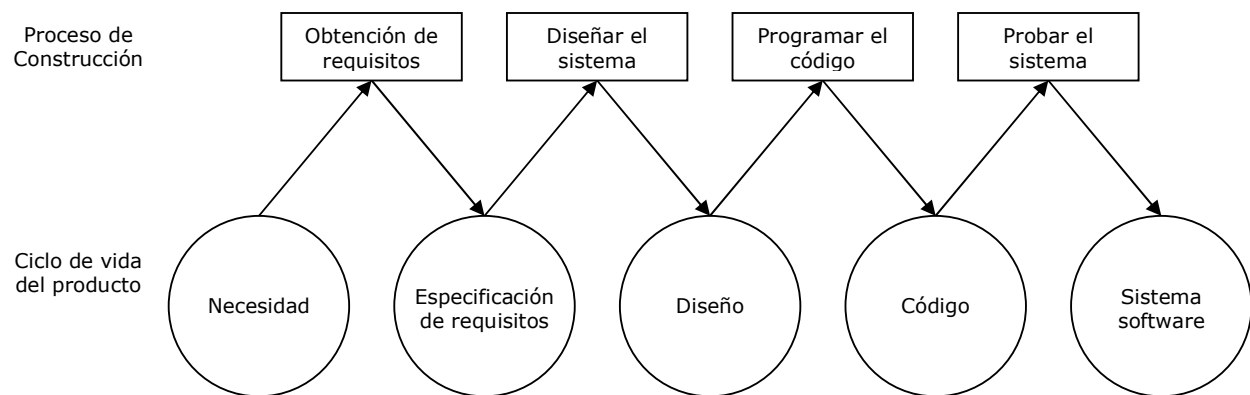
El mantenimiento consiste en la detección continuada de errores y su reparación. La ampliación, por su parte, se corresponde con la adición al sistema de nuevas capacidades. Estos dos procesos siguen, de hecho, un proceso completo como el visto hasta aquí, a pesar de que se recogen en una única etapa del proceso global para simplificar.

Proceso software frente a ciclo de vida

De todo lo visto hasta aquí, puede deducirse que el nombre **Proceso Software** se corresponde con la colección de actividades que comienza con la identificación de una necesidad y concluye con el retiro del software que satisface dicha necesidad. El proceso software más básico debe estar formado por las seis etapas vistas en la sección anterior. Sin embargo, en un proceso software, las actividades que lo constituyen deben estar interrelacionadas. Pueden existir más de una manera de interrelacionar las actividades, las distintas maneras representan distintas estrategias para cumplimentar la construcción de software.

Cámbiese ahora el enfoque usado hasta aquí. Se ha dicho que el problema planteado es la construcción de software, y que para resolver dicho problema se lleva a cabo un proceso de resolución, como cuando se trata cualquier otro problema. A dicho proceso de resolución se le ha dado el nombre de Proceso Software. Ahora bien, si el software obtenido tras el proceso es visto como el **producto** que *sale* del proceso, puede considerarse que cierta materia *entra* al proceso y se *transforma* a lo largo del mismo hasta obtener el producto deseado. De modo análogo a cómo una cadena de producción de coches puede verse como un proceso que obtiene coches como producto de salida. Para obtener los coches, el proceso se *alimenta* de piezas, planos, etc., que van transformando la entrada al proceso en el producto final (un coche). Dicho de otro modo, el proceso software puede verse como una cadena de tareas. Estas cadenas estructuran (o dictan) la transformación que van sufriendo las entidades computacionales (producto o solución) al pasar *a través* de una secuencia de acciones que forman cada actividad del proceso. Las cadenas de tareas son planes idealizados de qué acciones deben realizarse y en qué orden.

Con esta nueva perspectiva, se pueden establecer los *estados* por los que va pasando el producto en un proceso software: La *entrada* al proceso es una **necesidad** que, una vez estudiada, se convierte en una **especificación de requisitos** que, posteriormente, se transforma en un **diseño del sistema**, para pasar más adelante a ser un **código** y, finalmente, un **sistema software completo** e integrado. Este enfoque orientado al producto, focalizado en la transformación del producto, en lugar de en el proceso que lo transforma, es lo que se llama **Ciclo de Vida**. Es decir, el ciclo que el producto software sufre a lo largo de su vida, desde que *nace* (o se detecta la necesidad) hasta que *muere* (o se retira el sistema). La **Figura 3** muestra las relaciones entre Proceso y Ciclo de Vida.



El proceso software refleja cómo se usa la experiencia humana en la construcción de software y cómo se aplica a un dominio concreto. Es el ciclo de vida del software visto desde fuera. La estructura dentro de la que los ingenieros software deben operar.

CICLO DE VIDA

No existe un único **Modelo de Ciclo de Vida** que defina los estados por los que pasa cualquier producto software.

Un ciclo de vida debe:

- (1) Determinar el orden de las fases del Proceso Software
- (2) Establecer los criterios de transición para pasar de una fase a la siguiente

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

No existe un Modelo de Ciclo de Vida que sirva para cualquier proyecto, esto debe quedar claro. Cada proyecto debe seleccionar un ciclo de vida que sea el más adecuado para su caso. El ciclo de vida apropiado se elige en base a la cultura de la corporación, el deseo de asumir riesgos, el área de aplicación, la volatilidad de los requisitos, y hasta qué punto se entienden bien dichos requisitos.

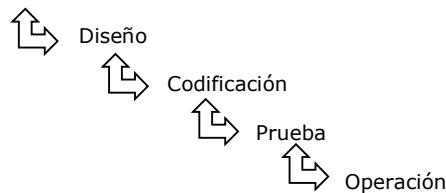
Algunos ciclos de vida

Modelo en cascada

Este modelo fue presentado por primera vez por Royce en 1970. Se representa, frecuentemente, como un simple modelo con forma de cascada de las etapas del software. En este modelo, la evolución del producto software procede a través de una secuencia ordenada de transiciones de una fase a la siguiente según un orden lineal. Tales modelos semejan una máquina de estados finitos para la descripción de la evolución del producto software. El modelo en cascada ha sido útil para ayudar a estructurar y gestionar grandes proyectos de desarrollo de software dentro de las organizaciones.

Este modelo permite iteraciones durante el desarrollo, ya sea dentro de un mismo estado, ya sea de un estado hacia otro anterior, como muestran las flechas ascendentes de la Figura. La mayor iteración se produce cuando una vez terminado el desarrollo y cuando se ha visto el software producido, se decide comenzar de nuevo y redefinir los requisitos del usuario.

Requisitos



El uso del modelo en cascada:

- (1) Obliga a especificar lo que el sistema debe hacer (o sea, definir los requisitos) antes de construir el sistema (esto es, diseñarlo).
- (2) Obliga a definir cómo van a interactuar los componentes (o sea, diseñar) antes de construir tales componentes (o sea codificar).
- (3) Permite al jefe del proyecto seguir y controlar los progresos de un modo más exacto. Esto le permite detectar y resolver las desviaciones sobre la planificación inicial.
- (4) Requiere que el proceso de desarrollo genere una serie de documentos que posteriormente pueden utilizarse para la validación y el mantenimiento del sistema.

Aún hoy en día se asume que:

- (1) Para que un proyecto tenga éxito, en cualquier caso, todos los estados señalados en el modelo en cascada deben ser desarrollados.
- (2) Cualquier desarrollo en diferente orden de los estados dará un producto de inferior calidad. Sin embargo, que se siga este orden no significa (como se pensó durante mucho tiempo) que se deba seguir la filosofía del ciclo de vida en cascada: realizar cada fase de principio a fin y no pasar a la siguiente hasta haber acabado completamente con la anterior.

Entre otras limitaciones que se argumentan para este modelo se pueden señalar las siguientes: El modelo en cascada asume que los requisitos de un sistema pueden ser congelados antes de comenzar el diseño.

Este es el punto más débil de este modelo, puesto que esta asunción es sólo cierta para un pequeño número de problemas: aquellos no complejos, bien definidos y bien comprendidos. Para el resto de los sistemas, es decir para la mayoría, las necesidades del usuario están en constante evolución. Por tanto, el sistema que se construye persigue un objetivo que no es fijo, sino que varía. Esta es una de las principales razones para los retrasos en las entregas del software: El equipo intenta que el sistema cumpla requisitos nuevos que han surgido durante el desarrollo y para los cuales el sistema no estará diseñado. La movilidad del objetivo que se persigue es también una de las razones para no satisfacer las expectativas de los usuarios.

Hoy por hoy se asume que intentar abarcar a través de una descripción escrita de requisitos, lo que el usuario piensa, cree y siente sobre el futuro software, jamás puede ser tan efectivo como un prototipo donde el usuario vea de lo que se habla (y más especialmente para la interface).

El fijar los requisitos al inicio del proyecto también suele traducirse en que el equipo de desarrollo emplee un gran esfuerzo en optimizar esa solución puntual. Eso lleva a que tal solución es difícil de modificar o aumentar, pues el

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

haber congelado los requisitos llevó a no pensar en los posteriores cambios que siempre existen en el usuario, su entorno, sus procedimientos y su organización.

Envía al cliente el primer producto solamente después de que se han consumido el 99% de los recursos para el desarrollo. Esto significa que la mayor parte del "feedback" del cliente sobre sus necesidades se obtiene una vez que se han consumido los recursos.

Sin embargo, el ciclo de vida en cascada tiene tres propiedades muy positivas:

- (1) Las etapas están organizadas de un modo lógico. Es decir, si una etapa no puede llevarse a cabo hasta que se hayan tomado ciertas decisiones de más alto nivel, debe esperar hasta que esas decisiones estén tomadas. Así, el diseño espera a los requisitos, el código espera a que el diseño esté terminado, etc.
- (2) Cada etapa incluye cierto proceso de revisión, y se necesita una aceptación del producto antes de que la salida de la etapa pueda usarse. Este ciclo de vida está organizado de modo que se pase el menor número de errores de una etapa a la siguiente.
- (3) El ciclo es iterativo. A pesar de que el flujo básico es de arriba hacia abajo, el ciclo de vida en cascada reconoce que los problemas encontrados en etapas inferiores afectan a las decisiones de las etapas superiores.

Construcción con refinamiento sucesivo o mejora iterativa

Las etapas que forman este ciclo de vida son las mismas que el modelo en cascada, y su realización sigue el mismo orden. Por tanto, tampoco se trata de un ciclo de vida nuevo, diferente del ciclo de vida en cascada. Se trata, simplemente de una recomendación orientada a mejorar el modelo de cascada.

Sin embargo, este modelo recomienda desarrollar los sistemas software a través de un refinamiento y mejora continuos desde las especificaciones de alto nivel del sistema hasta los componentes del código fuente. Es decir, este modelo asume que el producto generado en cada etapa no se produce de manera lineal, del principio al final de la etapa. Por el contrario, predica la generación de los productos de forma iterativa, mediante un proceso de refinamiento. Debido a la *marcha atrás* permitida en el modelo en cascada, que abre un camino desde una etapa hacia otra anterior, el refinamiento iterativo puede producirse también a nivel global de todas las etapas.

Modelo incremental

El primero que habló de este nuevo modelo fue Hirsch en 1985. Se trata de una filosofía radicalmente distinta del modelo de cascada. El desarrollo incremental es el proceso de construir una implementación parcial del sistema global y posteriormente ir aumentando la funcionalidad del sistema.

Esta aproximación reduce el gasto que se tiene antes de alcanzar cierta capacidad inicial. Además produce un sistema operacional más rápidamente; esto reduce la posibilidad de que las necesidades del usuario cambien durante el desarrollo.

Al seguir un tipo de desarrollo incremental, el software se construye de modo que deliberadamente sólo satisfaga unos pocos requisitos de todos los que tiene el usuario. Sin embargo, debe construirse de tal modo que facilite la incorporación de nuevos requisitos. Puede decirse, por tanto que el software así construido tiene una adaptabilidad mayor. Este tipo de aproximación tiene dos características:

- (1) Se reduce el tiempo de desarrollo inicial (hasta tener algo que funcione) debido al nivel reducido de funcionalidad.
- (2) Es más fácil de mejorar el software y, además, puede seguir mejorándose durante más tiempo.

En este modelo los puntos críticos del proyecto, o mojones a lo largo del camino de desarrollo, no son tanto los productos de cada etapa (especificaciones, diseño, código, etc.) sino las emisiones de incrementos en la capacidad del sistema. Los nuevos contenidos del sistema se determinan de la experiencia con las emisiones anteriores del sistema. Por tanto, el punto más crítico del sistema es la primera entrega: un software con capacidades suficientes como para servir para que el usuario se ejercite, para la evaluación y para evolucionar mejorando.

Los problemas que se le critican a este modelo de ciclo de vida son los siguientes:

- (1) A menudo, la primera entrega se ha optimizado para que tenga éxito como demostración y como software de exploración.
- (2) Las versiones iniciales no suelen prestar mucha atención a temas como: la seguridad, la tolerancia a fallos, el procesamiento distribuido, etc., centrándose en la funcionalidad del sistema y, quizás, en la interface. El usuario puede pensar que el resto de las capacidades del sistema, que no se han tenido en cuenta en la primera versión, se entreguen tan rápidamente como las primeras.
- (3) Cuando se usa este modelo de ciclo de vida, los ingenieros olvidan realizar una buena etapa de análisis que les lleve a comprender la necesidad del usuario.
- (4) Este modelo propone desarrollar sistemas produciendo en primer lugar las funciones esenciales de operación y, a continuación, proporcionar a los usuarios mejoras y versiones más capaces del sistema a intervalos

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

regulares. Este modelo combina el ciclo de vida clásico del software con mejoras iterativas a nivel del desarrollo del sistema global.

Prototipado

El modelo de desarrollo basado en prototipos tiene como objetivo contrarrestar el problema ya comentado del modelo en cascada: la congelación de requisitos mal comprendidos. La idea básica es que el prototipo ayude a comprender los requisitos del usuario. El prototipo debe incorporar un subconjunto de la función requerida al software, de manera que se puedan apreciar mejor las características y posibles problemas.

La construcción del prototipo sigue el ciclo de vida estándar, sólo que su tiempo de desarrollo será bastante más reducido, y no será muy rigurosa la aplicación de los estándares.

El problema del prototipo es la elección de las funciones que se desean incorporar, y cuáles son las que hay que dejar fuera, pues se corre el riesgo de incorporar características secundarias, y dejar de lado alguna característica importante. Una vez creado el prototipo, se le enseña al cliente, para que “juegue” con él durante un período de tiempo, y a partir de la experiencia aportar nuevas ideas, detectar fallos, etc.

Cuando se acaba la fase de análisis del prototipo, se refinan los requisitos del software, y a continuación se procede al comienzo del desarrollo a escala real. En realidad, el desarrollo principal se puede haber arrancado previamente, y avanzar en paralelo, esperando para un tirón definitivo a la revisión del prototipo.

El ciclo de vida clásico queda modificado de la siguiente manera por la introducción del uso de prototipos:

- (1) Análisis preliminar y especificación de requisitos
- (2) Diseño, desarrollo e implementación del prototipo
- (3) Prueba del prototipo
- (4) Refinamiento iterativo del prototipo
- (5) Refinamiento de las especificaciones de requisitos
- (6) Diseño e implementación del sistema final

Existen tres modelos derivados del uso de prototipos:

- (1) **Maqueta.** Aporta al usuario ejemplo visual de entradas y salidas. La diferencia con el anterior es que en los prototipos desechables se utilizan datos reales, mientras que las maquetas son formatos encadenados de entrada y salida con datos simples estáticos.
- (2) **Prototipo desechable.** Se usa para ayudar al cliente a identificar los requisitos de un nuevo sistema. En el prototipo se implantan sólo aquellos aspectos del sistema que se entienden mal o son desconocidos. El usuario, mediante el uso del prototipo, descubrirá esos aspectos o requisitos no captados. Todos los elementos del prototipo serán posteriormente desechados.
- (3) **Prototipo evolutivo.** Es un modelo de trabajo del sistema propuesto, fácilmente modificable y ampliable, que aporta a los usuarios una representación física de las partes claves del sistema antes de la implantación. Una vez definidos todos los requisitos, el prototipo evolucionará hacia el sistema final. En los prototipos evolutivos, se implantan aquellos requisitos y necesidades que son claramente entendidos, utilizando diseño y análisis en detalle así como datos reales.

La construcción de una maqueta no tiene mayor dificultad, pues se trata, simplemente, de implementar el dialogo entre la computadora y el usuario para confirmar las necesidades del usuario. Esto puede ser suficiente en problemas simples, pero para problemas más complejos el diálogo no es suficiente para comprender los requisitos. Es entonces cuando se opta por uno de los otros dos tipos de prototipo.

Prototipado desechable

El modelo de prototipado desechable pretende asegurar que el producto software que se está proponiendo cumple realmente las necesidades del usuario. La aproximación consiste en construir una implementación rápida, y no cuidada, parcial del sistema antes o durante la etapa de requisitos. El usuario utilizará este prototipo durante un tiempo y proporcionará retroalimentación a los desarrolladores sobre los puntos fuertes y débiles del prototipo. Esta retroalimentación se usa para modificar la especificación de requisitos de modo que refleje las verdaderas necesidades del usuario. Llegados a este punto, los desarrolladores pueden proceder con el diseño e implementación del sistema completo, teniendo la confianza que están construyendo el sistema adecuado (excepto para aquellos numerosos casos en que las necesidades del usuario evolucionen durante el desarrollo del sistema global). Una extensión de este modelo de ciclo de vida es usar más de un prototipo desechable.

Prototipado evolutivo

En este tipo de ciclo de vida (establecido por Jackson) los desarrolladores construyen una implementación parcial que satisface los requisitos conocidos. Entonces el usuario lo utiliza para llegar a comprender mejor la totalidad de requisitos que desea. Mientras el desarrollo incremental implica que se comprenden la mayor parte de los requisitos desde el principio y se elige implementarlos en subconjuntos de capacidad incremental o aumentada, el prototipo

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

evolutivo implica que no se conocen desde el principio todos los requisitos, y que se necesita experimentar con un sistema operacional para aprender cuáles son.

Nótese que en el prototipo desechable lo lógico es implementar sólo aquellos aspectos del sistema que se entienden mal, mientras que en el prototipo evolutivo lo lógico es empezar con aquellos aspectos que mejor se comprenden y seguir construyendo apoyados en los puntos fuertes, no en los débiles. En el caso de aplicaciones complejas, no es razonable esperar que la construcción del prototipo sea rápida. Por tanto la idea de prototipo de construcción rápida y poco cuidadosa se corresponde siempre con el desechable, nunca con el evolutivo. En el prototipo evolutivo incluso los requisitos de fiabilidad y rendimiento se implementan desde el principio. Esto es debido a que este tipo de requisitos, hoy por hoy, no se pueden incluir o ajustar con posterioridad, sino que para cumplirse deben contemplarse al mismo tiempo que se desarrollan las funcionalidades. No obstante, la idea del prototipo evolutivo se centra en funcionalidad y contenido.

Puede considerarse que, en cierto modo, el prototipado evolutivo es una extensión del desarrollo incremental. En aquel, el número y la frecuencia de prototipos operacionales son mayores que el número de versiones del sistema en la aproximación incremental. El énfasis, en el prototipado evolutivo, se pone en la evolución, de un modo más continuo, hacia una solución; en lugar de un número discreto de versiones del sistema.

Mediante tal aproximación, surge un prototipo inicial, antes que con el modelo clásico debido a lo reducido de sus funcionalidades, normalmente el prototipo demuestra la funcionalidad en aquellos requisitos bien entendidos (en contraste con el prototipo desechable, donde normalmente se implementan primero los aspectos peor entendidos). Este primer prototipo proporciona un marco para el resto del desarrollo. Cada sucesivo prototipo explora una nueva área de necesidades de usuario, además de refinar las funciones del prototipo anterior. Como resultado de este modo de desarrollo, la solución software evoluciona acercándose cada vez más a las necesidades del usuario. Pasado cierto tiempo, también el sistema software así construido deberá ser rehecho o sufrir una profunda reestructuración con el fin de seguir evolucionando.

Modelo en espiral

El **Modelo en Espiral** para el desarrollo de software representa un enfoque dirigido por el riesgo para el análisis y estructuración del proceso software. Fue presentado por primera vez por Böehm en 1986.

El enfoque incorpora métodos de proceso dirigidos por las especificaciones y por los prototipos. Esto se lleva a cabo representando ciclos de desarrollo iterativos en forma de espiral, denotando los ciclos internos del ciclo de vida análisis y prototipado precoz, y los externos, el modelo clásico. La dimensión radial indica los costes de desarrollo acumulativos y la angular el progreso hecho en cumplimentar cada desarrollo en espiral. El análisis de riesgos, que busca identificar situaciones que pueden causar el fracaso o sobrepasar el presupuesto o plazo, aparecen durante cada ciclo de la espiral. En cada ciclo, el análisis del riesgo representa groseramente la misma cantidad de desplazamiento angular, mientras que el volumen desplazado barrido denota crecimiento de los niveles de esfuerzo requeridos para el análisis del riesgo.

La primera ventaja del modelo en espiral es que su rango de opciones permite utilizar los modelos de proceso de construcción de software tradicionales, mientras su orientación al riesgo evita muchas dificultades. De hecho, en situaciones apropiadas, el modelo en espiral proporciona una combinación de los modelos existentes para un proyecto dado. Otras ventajas son:

- (1) Se presta atención a las opciones que permiten la reutilización de software existente.
- (2) Se centra en la eliminación de errores y alternativas poco atractivas.
- (3) No establece una diferenciación entre desarrollo de software y mantenimiento del sistema.
- (4) Proporciona un marco estable para desarrollos integrados hardware-software.

El Modelo en Espiral es un modelo de proceso de software evolutivo que acompaña la naturaleza interactiva de construcción de prototipos con los aspectos controlados y sistemáticos del Modelo en Cascada. Se proporciona el potencial para el desarrollo rápido de versiones incrementales del software.

Este modelo es un enfoque realista del desarrollo de sistemas y de software a gran escala. Como el software evoluciona, a medida que progresa el proceso, el desarrollador y el cliente comprenden y reaccionan mejor antes riesgos en cada uno de los niveles evolutivos.

El Modelo en Espiral utiliza la construcción de prototipos como mecanismo de reducción de riesgos y lo que es más importante, permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de la evolución del producto. Además, mantiene el enfoque sistemático de los pasos sugeridos por el ciclo de vida clásico pero lo incorpora al marco de trabajo interactivo que refleja de forma más realista el mundo real. Finalmente, demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto.

El Modelo en Espiral se divide en un número de actividades estructurales, también llamadas regiones de tareas. Entre ellas encontramos:

- (1) Comunicación con el cliente: Son las tareas requeridas para establecer la comunicación entre el desarrollador y el cliente.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

- (2) Planificación: Son las tareas requeridas para definir recursos, tiempo y otras informaciones relacionadas con el proyecto. A través de la iteración por esta etapa se producen ajustes en el plan del proyecto. El costo y la planificación se ajustan según la reacción ante la evaluación del cliente. Además, el gestor del proyecto ajusta el número planificado de iteraciones requeridas para completar el software.
- (3) Análisis de riesgos: Son las tareas requeridas para evaluar riesgos técnicos y de gestión.
- (4) Ingeniería: Son las tareas requeridas para construir una o más representaciones de la aplicación.
- (5) Construcción y adaptación: Son las tareas requeridas para construir, probar, instalar y proporcionar soporte al usuario.
- (6) Evaluación del cliente: Son las tareas requeridas para obtener la reacción del cliente según la evaluación de las representaciones del software creadas durante la etapa de ingeniería e implementada durante la etapa de instalación.

Si analizamos más en detalle el Modelo en Espiral comprenderemos que la dimensión radial representa el costo acumulativo incurrido en el logro de los pasos hasta la fecha; la dimensión angular representa el progreso hecho al terminar cada ciclo del espiral. (El modelo refleja el concepto subyacente que cada ciclo implica una progresión que trate la misma secuencia de pasos, para cada porción del producto y para cada uno de sus niveles de elaboración, desde un documento total de concepto de funcionamiento hasta la codificación de cada programa individual.)

Cada ciclo del espiral comienza con la identificación de:

- (1) Los objetivos de la porción del producto que es elaborado (el funcionamiento, funcionalidad, capacidad de adaptación al cambio, etc.);
- (2) Los medios alternativos de implementar esta porción del producto (diseño A, diseño B, reutilización, compra, etc.);
- (3) Y las restricciones impuestas ante el uso de las alternativas (costo, horario, interfaz, etc.).

El paso siguiente es evaluar las alternativas concernientes a los objetivos y restricciones. Con frecuencia, este proceso identificará las áreas de incertidumbre que son fuentes significativas de riesgo del proyecto. Si es así, el paso siguiente debe implicar la formulación de una estrategia rentable para resolver las fuentes de riesgo. Esto puede implicar el prototipado, simulación, benchmarking, chequeo de referencia, administración de cuestionarios al usuario, modelado analítico, o combinaciones de éstos y de otras técnicas de resolución de riesgos.

Una vez que se evalúen los riesgos, el paso siguiente es determinado por los riesgos restantes. Si los riesgos del funcionamiento o de la interfaz de usuario dominan fuertemente el desarrollo del programa o riesgos internos del control de interfaz, el paso siguiente puede ser un desarrollo evolutivo: un esfuerzo mínimo de especificar la naturaleza total del producto, un plan para el nivel siguiente de prototipado, y el desarrollo de un prototipo más detallado para continuar resolviendo los riesgos más importantes.

Si este prototipo es operacionalmente útil y bastante robusto servirá como base para la evolución futura del producto, los pasos subsecuentes de riesgos serán las series de desarrollo de prototipos evolutivos. En este caso, la opción de la escritura de especificaciones sería tratada pero no ejercitada. Así, las consideraciones del riesgo pueden conducir a un proyecto que pone solamente un subconjunto en ejecución de todos los pasos potenciales en el modelo.

Por otra parte, si los esfuerzos de prototipado anteriores han resuelto ya todos los riesgos del funcionamiento o de la interfaz de usuario, y los riesgos del desarrollo del programa o del control de interfaz están dominados, el paso siguiente es el acercamiento básico al Modelo en Cascada (conceptos de funcionamiento, requisitos del software, diseño preliminar, etc.), modificado para incorporarlo al desarrollo incremental. Cada nivel de la especificación del software es seguido por un paso de validación y preparación de los planes para el éxito del ciclo. En este caso, las opciones de prototipo, simulación, modelado, etc. se tratan pero no se ejercitan, conduciendo al uso de un diverso subconjunto de pasos.

El paso por el análisis de riesgos permite que el modelo se acomode a cualquier combinación apropiada orientada a la especificación, orientado al prototipo, orientada a la simulación, orientada a la transformación automática, o del otro acercamiento al desarrollo del software. En tales casos, la estrategia más apropiada es elegida considerando la magnitud relativa de los riesgos del programa y la eficacia relativa de las técnicas en la resolución de los riesgos. De una manera similar, las consideraciones de la administración de riesgo pueden determinar la cantidad de tiempo y el esfuerzo que se debe dedicar a las otras actividades tales del proyecto como planeamiento, administración de la configuración, garantía de calidad, verificación formal, y prueba.

Una característica importante del Modelo en Espiral, como con la mayoría de los otros modelos, es que cada ciclo es terminado por una revisión que implica todas las personas u organizaciones comprometidas con el producto. Esta revisión cubre todos los productos desarrollados durante el ciclo anterior, incluyendo los planes para el ciclo siguiente y los recursos requeridos para llevarlos hacia adelante. El objetivo principal de la revisión es asegurarse de que todas las partes en cuestión, estén conformes mutuamente para el acercamiento a la próxima fase.

Los planes por fases que tienen éxito pueden también incluir una partición del producto en incrementos para que el desarrollo o los componentes sucesivos sean convertidos por organizaciones individuales o personas. Para el último caso, visualice una serie de ciclos en espirales paralelos, uno para cada componente. Así, la revisión y aceptación pueden extenderse de un camino individual para el diseño del componente por parte de un solo programador a una

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

revisión importante de los requisitos que implican al desarrollador, el cliente, el usuario, y organizaciones de mantenimiento.


Ahora bien, ¿cuándo comienza y termina el espiral? Cuatro preguntas fundamentales se presentan en la consideración del Modelo en Espiral:

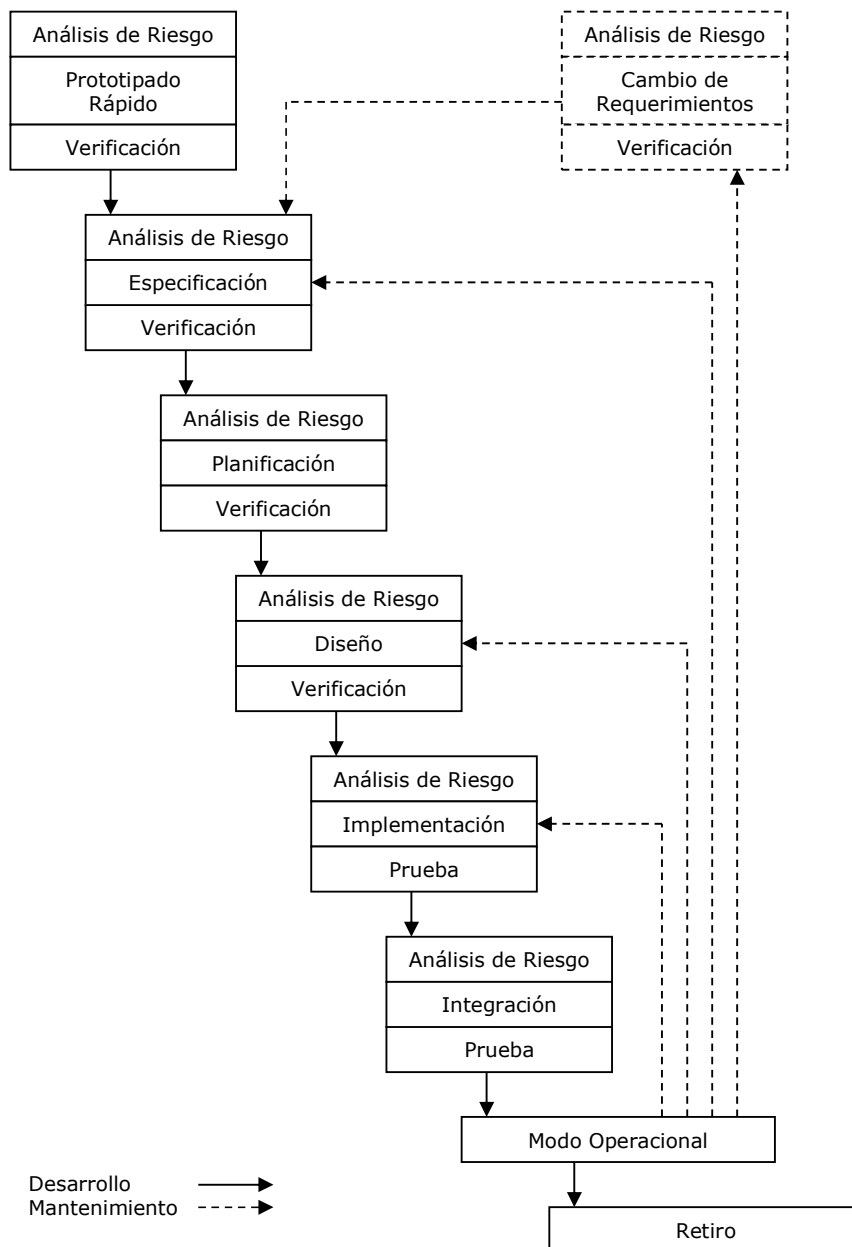
- (1) ¿Cómo el espiral consigue siempre comenzar?
- (2) ¿Cómo concluye el espiral cuando es apropiado terminar un proyecto temprano?
- (3) ¿Por qué el espiral termina tan precipitadamente?
- (4) ¿Qué sucede con el mantenimiento del software?

La respuesta a estas preguntas implica una observación, que el Modelo en Espiral se aplica igualmente bien a los esfuerzos de desarrollo como de mantenimiento. En cualquier caso, el espiral comienza con una hipótesis de una misión operacional particular (o el sistema de misiones) que se podría mejorar por un software. El proceso espiral entonces implica una prueba de esta hipótesis: en cualquier momento, si la hipótesis falla en la prueba, se termina el espiral. Si no, termina con la instalación del nuevo o modificado software, y la hipótesis es aprobada observando el efecto sobre la misión operacional. Generalmente, la experiencia con la misión operacional conduce a otras hipótesis sobre mejoras del software, y un nuevo espiral de mantenimiento se inicia para probar la hipótesis. La iniciación, la terminación, y la iteración de las tareas y de los productos de ciclos anteriores se definen así implícitas en el Modelo en Espiral.

Para concluir, cada iteración lleva consigo seis pasos:

- (1) Determinar objetivos, alternativas y límites (restricciones)
- (2) Identificar y resolver riesgos.
- (3) Evaluar las alternativas.
- (4) Generar las entregas de esta iteración, y comprobar que son correctas.
- (5) Planificar la siguiente iteración.
- (6) Establecer un enfoque para la siguiente iteración (si se decide ejecutarla)

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007



ANÁLISIS

¿Qué es el análisis?

En la Ingeniería del Software, el término “análisis” posee dos acepciones: Por una parte, se refiere a una determinada fase del proyecto de desarrollo; por otra, se refiere a un conjunto de productos que se originan como resultado de esta fase.

Se entenderá la actividad de análisis en el sentido de [Davis93] como “análisis del problema”, esto es, como una actividad conducente a identificar los conceptos relevantes del problema para poder, posteriormente, definir una solución software al mismo.

Esta decisión tiene ciertas implicaciones en el desarrollo del tema. Se podrían enunciar las siguientes:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.2 VIGENCIA: 19-09-2007
	APUNTE DE TEMAS DE REVISIÓN	

- (1) Al ser la actividad de análisis una actividad de identificación de conceptos de investigación, va a moverse siempre en un plano conceptual, lejos de cualquier aspecto de implementación del futuro sistema software. Sobre todo, el desarrollo que se va a realizar del tema va a ser ajeno al concepto de “requisito”, central en otras muchas aproximaciones al análisis. De hecho, no se va a hablar de requisitos en absoluto.
- (2) Muchas de los enfoques existentes para la realización de la actividad de análisis, como por ejemplo, el análisis estructurado, o el análisis orientado a objetos, tienen un marcado carácter constructivo. Por el contrario dichos enfoques se considerarán en este módulo de carácter descriptivo y, por lo tanto, sólo se tendrán en consideración, para su presentación, los conceptos más útiles para realizar la actividad de análisis, aunque la presentación pueda parecer incompleta.
- (3) Finalmente, todos los temas se presentarán independientemente de cualquier actividad de diseño, con lo que no se presentarán métodos que permitan derivar, a partir del análisis, productos de siguientes fases del proceso de desarrollo.

Objetivos del análisis

Cuando se inicia un proyecto de software pueden darse, fundamentalmente, dos situaciones:

- (1) Que el proyecto de desarrollo verse sobre un tema conocido, ya habitual para el equipo de trabajo.
- (2) Que el proyecto tenga como objetivo desarrollar un software novedoso, o que el entorno de trabajo y los conceptos manejados sean desconocidos.

Un ejemplo de un proyecto del primer tipo sería el de una empresa que ejerciera su actividad en un determinado sector y que posea un departamento de desarrollo dedicado, en exclusiva, a desarrollar software para uso interno de la empresa. Es previsible, como ocurre efectivamente en la mayoría de los casos, que concurran varias circunstancias que favorezcan la estabilidad del proyecto:

- (1) El entorno de trabajo estable, esto es, no ocurren cambios bruscos en la actividad ejercida por la empresa, ni en el mercado, ni en los potenciales clientes.
- (2) El equipo de trabajo es prácticamente el mismo en todos los proyectos, esto es, la rotación del personal es mínima o inexistente.
- (3) Los sistemas software desarrollados son variaciones, en distinto grado de amplitud, de otros sistemas ya desarrollados con anterioridad.
- (4) Las plataformas hardware y software cambian con muy poca frecuencia, y los cambios que se producen son, fundamentalmente, cambios en la versión de un producto previamente instalado y en uso.
- (5) Es evidente que, en este caso, el equipo de desarrollo ya conozca de antemano muchos de los parámetros del desarrollo: funcionalidad a entregar, características no funcionales del producto, factores de riesgo, etc.

En el segundo caso, probablemente todos los factores anteriores no están fijados o son desconocidos. Pues bien, la actividad de análisis es una estimable ayuda en los proyectos del primer tipo, y es un factor crucial en los del segundo, ya que su objetivo es, en palabras de [Davis93]:

“El análisis es la actividad que incluye aprender acerca del problema a resolver [...], entender las necesidades de los potenciales usuarios, tratar de identificar quién es realmente el usuario y entender todas las restricciones a la solución.”

Dos palabras son claves en la cita anterior: *aprender* y *entender*. El análisis es una actividad que pretende proporcionar al profesional de la Ingeniería del Software el conocimiento necesario para solucionar los problemas que surgen en un dominio determinado y que, habitualmente, es ajeno a él mismo. El conocimiento a adquirir es, por lo tanto, extraño, y sólo puede obtenerse mediante un aprendizaje *in situ*, esto es, mediante una inmersión en el dominio del problema, dominio perteneciente a los clientes y/o usuarios que sufren el problema que el profesional debe solucionar.

El análisis sirve, por lo tanto, para identificar los conceptos del problema a resolver, y entender las restricciones que debe cumplir cualquier solución software que le sea aplicable. Es a partir del conocimiento adquirido en esta fase de lo que se sirve el profesional de la Ingeniería del Software para derivar una solución.

¿Quién debe realizar el análisis?

La persona que realiza el análisis, al que típicamente se denomina *analista*, no tiene por qué ser un experto informático. Su labor, por el contrario, debe ser inquisidora, de investigación y aprendizaje, con el objetivo de aprender lo más posible acerca del dominio del problema.

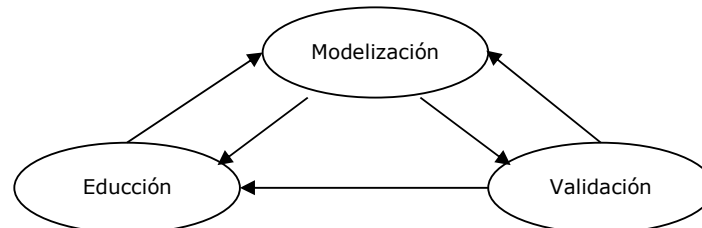
Por último, el analista debe poseer especiales habilidades de comunicación, ya que debe realizar una labor de puente entre los dos mundos presentes en el análisis: el mundo técnico de los desarrolladores, y el mundo informal de los clientes y/o usuarios. Debe indicarse, adicionalmente, que los desarrolladores, clientes y usuarios acostumbran a denominarse “participantes” (stakeholders) de la actividad de análisis.

¿Cuáles son las tareas de análisis?

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

Las tareas a realizar durante la actividad de análisis no están muy definidas. No obstante, en los últimos años se ha desarrollado una gran actividad investigadora en un campo denominado "Ingeniería de Requisitos", el cual incluye de forma natural al análisis. El resultado de esta actividad investigadora es la identificación de dos aproximaciones distintas para la realización de la actividad de análisis. Dichas aproximaciones son las siguientes:

La aproximación clásica, expuesta en trabajos como los de [Loucopoulos95] y [Jalote96] consta de tres tareas.



La descripción de las tareas es la siguiente:

Educción: Tiene como objetivo adquirir el conocimiento del dominio de los clientes y/o usuarios, de tal forma que sea posible identificar los conceptos, relaciones y funciones más relevantes.

Esta tarea tiene características, por lo tanto, de investigación, y es difícilmente formalizable, dada la variedad de conocimientos dependientes del dominio que puede ser necesario adquirir. En Ingeniería del Software, existe una serie de técnicas básicas para realizar esta tarea, como son las entrevistas (ya sean abiertas o estructuradas), los cuestionarios o el análisis de documentos, así como algunas más elaboradas como el prototipado.

Modelización: Consiste en representar, mediante la utilización de "Modelos Conceptuales", los conocimientos adquiridos en la tarea anterior. Los modelos conceptuales son mecanismos de representación que permiten registrar, de una forma sencilla, los conocimientos adquiridos durante la educción con el fin de facilitar su comprensión y permitir su comunicación entre todos los participantes en la actividad de análisis (clientes, usuarios y analistas).

Validación: Consiste en verificar la exactitud de los conocimientos adquiridos, ya que no existe ninguna razón para que el analista sea infalible en su actuación.

Las técnicas de validación al nivel de análisis, a diferencia de en otras actividades del proceso de desarrollo, son muy débiles, existiendo prácticamente una única técnica: la revisión con los clientes y usuarios.

La aproximación más moderna, propuesta por el SWEBOK (Software Engineering Body of Knowledge) y que de momento está en proceso de revisión y refinamiento.

Esta aproximación, que será el marco de referencia dentro de unos años, propone un conjunto de cuatro tareas para la fase completa de "ingeniería de requisitos", de la cual el análisis forma parte.

Es evidente que las tareas que plantea el SWEBOK deben considerarse una ampliación (y no demasiado excesiva) de las planteadas por la aproximación clásica. De hecho, la primera tarea (educción), y la última tarea (validación) propuestas en el SWEBOK coinciden con la aproximación clásica. Las dos tareas restantes son las siguientes:

Análisis y negociación: En esta tarea se engloban tres subtareas bastante homogéneas, aunque de diferente importancia para el análisis, tal y como se plantea en el presente módulo:

- (1) Clasificación de requisitos: Tiene como objetivo clasificar, para facilitar su comprensión y su utilización en posteriores actividades del desarrollo, los requisitos identificados durante la tarea de educción.
- (2) Modelado Conceptual: Tiene los mismos objetivos que en la aproximación clásica, con la salvedad de que, en lugar de estar trabajando con conocimiento del dominio, se está trabajando con requisitos del sistema.
- (3) Negociación: Tiene como objetivo solucionar los problemas que puedan surgir debido a conflictos entre requisitos.

Documentación de requisitos: Tiene como objetivo estabilizar los requisitos identificados, y sin conflictos, en un documento que sirva de referencia para todo el proceso de desarrollo.

El papel del conocimiento previo

El análisis se caracteriza, como ya se ha indicado anteriormente, por ser una actividad que se desarrolla entre dos dominios, el del cliente y/o usuario y el del desarrollador. Por lo tanto, el analista está obligado a realizar una labor de puente, de comunicador entre dos mundos habitualmente muy distantes.

Para realizar dicha labor de puente, el analista necesita, como mínimo, comprender el mundo del desarrollador. Esta es una precondition, no eludible, que define un perfil mínimo para el analista, y determina uno de los tipos de conocimiento que el analista debe poseer: el conocimiento técnico. El conocimiento técnico es necesario, dado que la

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

salida del análisis debe servir, en primera instancia, para confeccionar una Especificación de Requisitos del Software que guíe todo el proceso de desarrollo posterior. Para realizar efectivamente la tarea de Especificación de Requisitos, el analista, sea o no el que realice la Especificación, debe comprender el propósito de ésta, los principios que guían su confección, las técnicas de validación y los aspectos de trazabilidad, gestión de cambios y gestión de calidad.

Sin embargo, la utilidad del conocimiento técnico es nula sin conocimiento específico del dominio del usuario y/o cliente. Dicho de otra forma, es casi imposible resolver un problema si no se conoce el problema particular y la forma particular de resolverlo. No existe, en la actualidad, ninguna clasificación de dominios de cliente y/o usuario, por lo que es difícil realizar una descripción de los mismos. No obstante, existen una serie de características que, habitualmente, se repiten en los distintos dominios de clientes y/o usuarios:

Inconsistencia: La información que obtiene el analista al sumergirse en el dominio de los usuarios y/o clientes no es coherente. Habitualmente, el analista debe consultar diversas fuentes de información, y los datos que obtiene son a menudo contradictorios.

Incompletitud: La información relevante para el análisis no acostumbra a estar centralizada, sino dispersa en multitud de fuentes. El trabajo del analista se puede comparar a menudo con montar un "puzzle", pero sin saber dónde están las piezas.

Ambigüedad: La información habitualmente no es clara, pudiendo entenderse en varios sentidos sin poder, a priori, adjudicar uno de ellos unívocamente.

Estos tres problemas no aparecen nunca aislados, sino fuertemente entrelazados, y suponen una fuente de riesgo. La única forma de evitarlos es el conocimiento previo.

Por conocimiento previo debe entenderse conocimiento propio del dominio o de dominios similares. Cuando el analista dispone de dicho conocimiento, aunque no sea ni mucho menos completo, puede enfrentarse a la incompletitud, inconsistencia y ambigüedad con más posibilidades de éxito.

Alcance del análisis

Uno de los problemas de la actividad de análisis es que se sabe cuándo empieza, pero es muy difícil saber cuando acaba. Dicho de otra forma; cuando empieza un proyecto de desarrollo, se sabe que debe comenzar con la actividad de análisis, pero lo que se ignora es cuándo terminará ésta.

El problema de cuándo termina la actividad de análisis está relacionado (de hecho se deriva) con determinar qué es análisis y qué es diseño. Esta pregunta no es nueva, pero todavía no ha sido contestada adecuadamente, y es probable que nunca lo sea. En los siguientes párrafos se intentará aportar una (nueva) respuesta, la cual será la base para determinar el alcance de la actividad de análisis.

Análisis vs. Diseño

La actividad de análisis es plenamente conceptual, esto es, su propósito fundamental es conseguir el entendimiento del dominio de los clientes y/o usuarios con lo que consigue *penetrar* en el dominio y alcanzar una comprensión (adecuada) del mismo.

Desgraciadamente, el diseño tiene casi los mismos objetivos, pero referentes a un sistema basado en computadora, y no al dominio de los clientes y/o usuarios. Adicionalmente, las herramientas conceptuales de la actividad de diseño son, en gran medida, similares a las utilizadas durante el análisis.

Por esta razón, análisis y diseño a veces se confunden y sus objetivos se mezclan. Lo más normal es que, durante el análisis, el analista comience intentando entender el dominio y, a medida que lo va comprendiendo, vaya pergeñando la solución software; esto es, el analista comienza haciendo un análisis para, finalmente, avanzar hacia el diseño.

"El análisis debe terminar cuando los conceptos del dominio están lo suficientemente claros como para que el analista sea capaz de idear una solución software para el problema bajo estudio"

No obstante, esta regla posee casi tanta indefinición como la propia indefinición del alcance del análisis. Existen dos problemas asociados a la misma:

- (1) El analista, con frecuencia, no es consciente de que está pergeñando una solución, en lugar de intentar comprender los conceptos y reglas que gobiernan el dominio.
- (2) El análisis tiene muchas facetas: se investigan varios subdominios a la vez, y desde distintos puntos de vista. Así, se puede comprender un subdominio o una serie de facetas de varios subdominios, y trabajar de una forma mixta, analizando algunas cosas y diseñando otras.

La solución, por lo tanto, no es sencilla, pero la regla anterior es la única posible. Si alguna faceta está lo suficientemente bien comprendida que ya es posible diseñar una solución, es mejor dejar de trabajar sobre ella y concentrar la atención y esfuerzos sobre las restantes.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

DISEÑO

En el momento en que dicha actividad comienza tenemos una serie de requisitos, que reflejan lo que se desea que el sistema realice, y nuestra misión será generar un modelo del sistema software que implementará los requisitos de partida. Por supuesto, no va a existir un único diseño válido; distintos diseñadores pueden construir distintos diseños y ser todos válidos. Asimismo, dependiendo del contexto del futuro sistema, un diseño puede ser mejor que otro.

Un buen diseño es la clave para una ingeniería efectiva pero no es posible formalizar el proceso de diseño en una ingeniería, ya que por desgracia, en muchas organizaciones el diseño del software es aún un proceso ad-hoc. Dado un conjunto de requisitos, normalmente en lenguaje natural, se prepara un diseño informal. Se comienza a codificar y el diseño se va modificando a medida que el sistema es implementado.

Diseño e Ingeniería del Software

El proceso de diseño consiste en traducir los requisitos a una representación del software que va a ser implementado. Al fin y al cabo, un diseño software es un modelo del futuro sistema, que ha de incluir los componentes, sus conexiones, las interfaces, los algoritmos y estructuras de datos, así como una serie de procedimientos que definen el uso que tendrá el futuro sistema.

Para satisfacer a los distintos usuarios, se suelen usar tres tipos de representaciones:

- (1) **Notaciones gráficas.** Se usan para mostrar las relaciones entre los componentes que conforman el diseño y para relacionar el diseño con el sistema del mundo real que está modelizando. Una vista gráfica de un diseño es una vista abstracta y es útil sobre todo para dar una vista genérica del sistema.
- (2) **Lenguajes de descripción de programas (PDL).** Estos lenguajes usan construcciones de control y estructuras basadas en las usadas por los lenguajes de programación, aunque también usan texto explicativo. La diferencia entre los lenguajes de descripción de programas y los de programación es que permiten al diseñador expresarse en lugar de centrarse en detalles acerca de cómo se implementará el diseño.
- (3) **Texto informal.** Gran parte de la información asociada a un diseño no se puede expresar formalmente. La explicación acerca de la razón de ser del diseño o consideraciones acerca de los requisitos no funcionales deben expresarse usando el lenguaje natural.

Teóricamente, deberían usarse todas estas notaciones a la hora de describir un diseño. La descripción gráfica debería usarse para describir la arquitectura y el diseño lógico de los datos, complementándose con la razón de ser del diseño en texto descriptivo informal o formal. El diseño de la interfaz, el diseño detallado de las estructuras de datos y el diseño de los algoritmos se hacen mejor usando los PDL. Se pueden incluir otras explicaciones en forma de comentarios.

Distintos autores hablan de la existencia de distintos niveles de diseño, aunque todos coinciden en la existencia de dos niveles a grandes rasgos:

- (1) **Diseño preliminar o diseño de la arquitectura.** Transforma los requisitos en una arquitectura software del sistema. La arquitectura software de un sistema es la estructura del mismo, y comprende los componentes software (módulos, procesos, etc.), las propiedades externamente visibles de dichos componentes (asunciones que otros componentes pueden hacer de un componente dado, como los servicios que provee, las características de rendimiento, manejo de errores, uso de recursos compartidos, etc.), y las relaciones entre ellos. La arquitectura incluye información acerca de cómo interactúan unos componentes con otros, pero a su vez suprime detalles de los componentes que no afectan a: cómo usan, son usados, se relacionan con o interactúan con otros componentes. En resumen, todos aquellos detalles que tienen que ver únicamente con la implementación interna no pertenecen al diseño de la arquitectura.
- (2) **Diseño de bajo nivel.** Refina el diseño preliminar, y comprende el diseño de estructuras de datos y de algoritmos.

El diseño consta de una serie de actividades como la elección de la/s arquitectura/s a usar, el diseño de la estructura del sistema, el diseño de los interfaces, el diseño de los componentes, el diseño de las estructuras de datos, de algoritmos, etc.

Conceptos Fundamentales del Diseño de Software

Una vez que se ha visto lo que es el diseño de software, a continuación se mostrarán aquellos conceptos fundamentales en los que se basa:

- (1) Modularidad.
- (2) Abstracción.
- (3) Ocultación de la información.

El diseño siempre conlleva una descomposición del sistema software a desarrollar: inicialmente se comienza con una descripción a alto nivel de los elementos principales del sistema y posteriormente se van elaborando descripciones cada vez a más bajo nivel de cuál será la forma del sistema y sus funciones.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

Independientemente de la heurística a seguir a la hora de realizar el diseño, la descomposición identificará cada una de las partes que componen el diseño del sistema, llamadas componentes. Se dice que un sistema es modular cuando cada actividad del sistema la lleva a cabo un único componente, y cuando la interfaz de cada componente (entradas y salidas) está bien definida.

Los sistemas modulares tienen también la ventaja de que están organizados en jerarquías como consecuencia de la descomposición. Cada uno de los niveles de descomposición representa un nivel de abstracción. La abstracción permite concentrarse en un problema a un nivel de generalización independientemente de los detalles de nivel inferior.

De igual modo, la modularidad esconde detalles. Según Parnas, los componentes esconden los detalles internos y de procesamiento unos a otros. Una de las ventajas de la ocultación de la información es que cada componente esconde una decisión de diseño a los otros. De este modo, si ciertas decisiones de diseño son propensas a cambiar, el diseño puede permanecer intacto mientras sólo el diseño de un componente cambia.

La combinación de estas tres características (modularidad, abstracción y ocultación de la información) nos ofrece distintas vistas del sistema. Los componentes de más alto nivel nos dan la oportunidad de ver el diseño como un todo, escondiendo detalles que no son de interés (y que por tanto nos distraerían) mientras que los de más bajo nivel nos permiten concentrarnos en un único componente, olvidándonos de la existencia del resto. Esto se refleja en la posibilidad de diseñar distintas partes de un sistema de distintos modos.

Contrariamente a lo que piensa la gente, dividir un problema en partes no hace de un problema complejo un conjunto de simples, pero permite aislar aquellas partes del problema que son las más difíciles de tratar; los niveles de abstracción permiten entender el problema a mayores niveles de detalle.

Proceso de Diseño de Software

Tal y como se ha dicho anteriormente, durante el diseño se busca obtener un modelo que represente el futuro sistema. No obstante, esto no es tarea fácil, y durante esta fase deberán realizarse distintas actividades:

- (1) Selección de la arquitectura global del sistema.
- (2) Descomposición en subsistemas.
- (3) Identificación de concurrencia.
- (4) Control de recursos compartidos.
- (5) Gestión de los datos.
- (6) Diseño de las interfaces.
- (7) Identificación y manejo de excepciones.
- (8) Selección de algoritmos y estructuras de datos.

En numerosas ocasiones, la arquitectura general de un sistema se elige por similitud con otros sistemas previamente desarrollados. Es necesario elegir el/los tipo/s de arquitectura/s que se utilizarán a la hora de diseñar el software, ya que no todos son útiles para diseñar todos los sistemas software. Asimismo, es muy normal el diseñar sistemas resultantes de la combinación de varios tipos de arquitecturas.

Una de las principales características del diseño es la modularidad. Para conseguir la modularidad es necesario descomponer el sistema hasta conseguir unidades elementales. La forma de descomponer el sistema vendrá impuesta normalmente por la arquitectura global elegida (los criterios de descomposición son distintos), aunque en otros casos podrá ser al revés (elegiremos una arquitectura por la forma que tiene de realizar la descomposición).

Muchas de las tareas que tiene que realizar un sistema software pueden ser conceptualmente concurrentes (es decir, que en teoría podrían realizarse a la vez puesto que son independientes). Sin embargo, y a efectos de diseño, se debe tener en cuenta las limitaciones del hardware y buscar un compromiso entre el coste del mismo y la mejora esperada en el rendimiento de la aplicación. Una vez que se ha decidido la concurrencia de los elementos de un sistema, es necesario establecer además los mecanismos que posibiliten su implementación.

No sólo va a haber que controlar el acceso al hardware (procesadores, disquetes, etc.), sino que también habrá que tener en cuenta el acceso a entidades lógicas como son los ficheros, las bases de datos, etc. Todos los recursos globales del sistema deben ser adecuadamente gestionados para garantizar el acceso por todas aquellas entidades que estén autorizadas a usarlos.


Los datos que manejará el sistema software pueden ser almacenados de distintos modos.

Cada tipo de almacén tiene sus ventajas y sus desventajas. Algunos de los almacenes más utilizados son los ficheros y las bases de datos.

El diseño de la interfaz se centra no sólo en el diseño de las interfaces de usuario, sino también en el diseño de interfaces entre los distintos componentes del software y también las interfaces entre el software y otras entidades no humanas (hardware, otro software, etc.).

Calidad del Diseño de Software

Lo que tiene que quedar claro es:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

- (1) El diseño de la arquitectura es crítico a la hora de conseguir los atributos de interés en un sistema, y estos atributos deberían ser estudiados y evaluados a nivel de diseño.
- (2) Algunos atributos no son sensibles a la arquitectura y por tanto su estudio a nivel de diseño no tiene sentido.

Hay dos tipos de atributos de calidad que se pueden medir en un sistema (y en concreto a nivel de diseño).

- (1) **Observables vía ejecución.** Son aquellos cuyo valor se obtiene observando el sistema en ejecución. Miden cosas como: ¿Hasta qué punto satisface el sistema los requisitos de comportamiento? ¿Proporciona los resultados requeridos? ¿Los proporciona en un período de tiempo razonable? ¿Son los resultados correctos o están dentro del rango requerido de precisión? ¿Funciona correctamente el sistema cuando se conecta con otros?
- (2) **No observables vía ejecución.** Son aquellos cuyo valor se obtiene estudiando el sistema software en reposo. Miden cosas como: ¿Hasta qué punto es fácil integrar, probar y modificar el sistema? ¿Cuál fue su coste de desarrollo? ¿Cuánto se tardó en desarrollarlo?

El conocer cosas acerca de la primera categoría no nos dice nada acerca de la segunda.

Vamos a examinar las siguientes clases de atributos:

- (1) Atributos propios del diseño.
- (2) Atributos del software observables a través de su ejecución y que conciernen al diseño.
- (3) Atributos del software que no se pueden observar a través de la ejecución y que conciernen al diseño.

Atributos de un Buen Diseño

A la hora de realizar un diseño se deberá tener en cuenta: el acoplamiento, la cohesión, la adaptabilidad y la entendibilidad. A continuación se verá cada uno de ellos con mayor grado de detalle.

Acoplamiento

Un concepto importante en diseño es la independencia. Se dice que dos componentes son totalmente independientes si cada uno de ellos puede funcionar sin la presencia del otro, lo que implica que no hay interconexiones entre ellos. De modo genérico, cuantas más conexiones haya entre componentes, menos independientes serán.

La cuestión clave cuando se habla del acoplamiento es: ¿cuánto ha de saber un componente acerca de otro para poder entenderlo? Esto precisamente es lo que se denomina acoplamiento, y mide la fuerza de la conexión entre dos componentes. El acoplamiento como concepto abstracto se puede definir como la probabilidad de que a la hora de codificar, depurar o modificar un componente, un programador tenga que tener en cuenta algo acerca de otro.

El acoplamiento depende de varias cosas:

- (1) Las referencias que hace un componente a otro. Por ejemplo, el componente A puede llamar al componente B, de tal modo que A depende de B para completar su función o proceso.
- (2) Cantidad de datos pasados de un componente a otro. Por ejemplo, el componente A puede pasar un parámetro, el contenido de un vector, o un bloque de datos al componente B.
- (3) Cantidad de control que un componente tiene sobre otro. Por ejemplo, el componente A puede pasar un flag de control al componente B. El valor del flag le dice al componente B el estado de algún recurso o subsistema, si ha de invocar algún proceso o qué proceso ha de invocar.
- (4) El grado de complejidad de la interfaz entre componentes. Por ejemplo, el componente A pasa un parámetro al componente B, después de lo cual B se puede ejecutar. Pero los componentes C y D intercambian valores antes de que D pueda completar su ejecución, por tanto la interfaz entre A y B es menos compleja que la de C y D.

El objetivo es mantener el grado de acoplamiento tan bajo como sea posible para minimizar la dependencia entre componentes. De este modo solo unos pocos componentes serán afectados al cambiar uno.

Existen distintos tipos de acoplamiento, siendo algunos menos deseables que otros.

- (1) **Acoplamiento de contenido:** ocurre cuando una unidad de software necesita acceder a una parte de otra unidad de software.
- (2) **Acoplamiento común:** dos unidades de software acceden a un mismo recurso común, generalmente memoria compartida, una variable global o un fichero.

Si bien se reduce el grado de acoplamiento al organizar el diseño de tal modo que los datos sean accesibles desde un almacén de datos común, la dependencia aún existirá, ya que un cambio en los datos comunes significa identificar todos los componentes que acceden a los datos para evaluar el efecto del cambio.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

- (3) **Acoplamiento externo:** las unidades de software están ligadas a componentes externos, como por ejemplo dispositivos de entrada/salida, protocolos de comunicaciones, etc.
- (4) **Acoplamiento normal:** una unidad de software llama a otra de un nivel inferior y tan solo intercambian datos, por ejemplo, parámetros de entrada/salida. Dentro de este tipo de acoplamiento podemos encontrarnos 3 subtipos, dependiendo de los datos que intercambien las unidades de software:
- Acoplamiento de control:* los datos que se intercambian entre las unidades de software son controles. Debido a que en este subtipo una unidad controla la ejecución de otra, no es un buen acoplamiento, ya que impide que sean totalmente independientes.


El componente controlado no puede funcionar sin las instrucciones del que controla. En un diseño con acoplamiento de control se tiene la ventaja de que cada componente lleva a cabo una única función o ejecuta un proceso. Esta restricción minimiza la cantidad de información de control que se debe pasar de un componente a otro y localiza el control en un conjunto fijo de parámetros que forma una interfaz bien definida.
 - Acoplamiento de marca o por estampado:* las unidades de software se pasan datos con estructura de registro. No es muy deseable si la unidad receptora sólo requiere parte de los datos que se le pasan, a su vez, los valores de los datos, formato y organización de los componentes que interactúan debe coincidir.
 - Acoplamiento de datos:* las unidades de software se comunican mediante parámetros donde solo se pasan datos.

Cohesión

La forma en la cual se divide físicamente un sistema en componentes puede afectar de modo significativo a la complejidad estructural del sistema resultante, y también al número total de referencias entre componentes. Ahora estamos considerando la cohesión de cada componente de forma independiente, o el grado de relación entre sus elementos internos. La cohesión de un componente puede ser definida como el cemento que une o junta a los elementos de proceso de un componente. Es uno de los factores cruciales en diseño, y es constituyente fundamental de modularidad efectiva.

Claramente, cohesión y acoplamiento están interrelacionados. Cuanto mayor es la cohesión de componentes individuales del sistema, menor será el acoplamiento entre ellos.

- (1) **Cohesión casual o coincidente:** los elementos de la unidad de software contribuyen a las actividades relacionándose mutuamente de una manera poco significativa. Cuando ocurre esto, se pueden encontrar funciones, procesos o datos no relacionados dentro del mismo componente por razones de conveniencia. Este tipo de cohesión viola el principio de independencia y de caja negra de las unidades de software, por lo tanto debe evitarse.
- (2) **Cohesión lógica:** cuando las unidades de software agrupadas realizan un trabajo en una misma categoría lógica, pero no necesariamente tienen relación entre sí. Por ejemplo, bibliotecas de funciones matemáticas, sólo se agrupan porque realizan cálculos matemáticos, pero no tienen necesariamente relación entre ellas, otro ejemplo sería un componente que lee todo tipo de entradas, CD, disco, red, etc., independientemente desde dónde proceda dicha entrada o cómo sea usada; la entrada es el pegamento que mantiene a este componente unido. A pesar de que este tipo de cohesión es más razonable que la casual, los elementos de un componente con cohesión lógica no están relacionados funcionalmente.
- (3) **Cohesión temporal:** los elementos de la unidad de software están implicados en actividades relacionadas con el tiempo. En otras palabras, se agrupan unidades de software que tienen que ejecutarse más o menos en el mismo período de tiempo, sin que haya otro tipo de relación entre ellas. Por ejemplo, un componente que sirve para inicializar un sistema o un conjunto de variables. Los componentes con cohesión lógica y temporal son difíciles de cambiar y deben evitarse.
- (4) **Cohesión procedimental:** la unidad de software tiene una serie de funciones relacionadas por un procedimiento efectuado por el código. A menudo, las funciones de un sistema deben llevarse a cabo en un orden determinado por tal se agrupan funciones en un componente para asegurar este orden.
- (5) **Cohesión de comunicación o de datos:** la unidad de software realiza actividades paralelas usando los mismos datos de entrada y salida. En otras palabras, cuando todas las unidades agrupadas trabajan sobre el mismo conjunto de datos. Por ejemplo, a veces datos que no están relacionados se buscan juntos porque la búsqueda se puede hacer en un sólo acceso. Este tipo de cohesión a menudo destruye la modularidad y la independencia funcional del diseño.
- (6) **Cohesión secuencial:** una unidad de software realiza distintas tareas en secuencia, de forma que las entradas de cada tarea son las salidas de la tarea anterior. En otras palabras, se agrupan las unidades que

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.2 VIGENCIA: 19-09-2007
APUNTE DE TEMAS DE REVISIÓN		

cumplen que los resultados o salidas que produce una sirven como entrada para que la próxima continúe trabajando. Como el componente no está construido basándose en relaciones funcionales, es posible que no restrinja todo el procesamiento relativo a una función. A su vez, es similar a la procedimental salvo que en este último caso se incluye un paso de control que no es requerido en el secuencial.

- (7) **Cohesión funcional:** los elementos de la unidad de software están relacionados en el desarrollo de una única función. Es decir, las unidades de software trabajan juntas con un mismo fin. En general, es el criterio de agrupación más deseable. Probablemente haya entre las unidades un acoplamiento relativamente alto, por lo tanto es conveniente que estén juntas. Un componente con cohesión funcional, no sólo realiza la función para la que se diseñó, sino que realiza únicamente esa función y nada más.

Adaptabilidad

La adaptabilidad juega un papel crucial en el mantenimiento. En la adaptabilidad influyen cosas como el acoplamiento y la documentación, que debe ser buena además de entendible y consistente con la implementación, la cual debe estar escrita de un modo entendible.

Un diseño adaptable debe tener mucha visibilidad, lo que significa que la relación entre los distintos niveles de diseño debe ser lo más directa posible.

También debería ser fácil incorporar a la documentación los cambios hechos en el diseño.

Esto debe hacerse siempre, ya que si no se podría dar el caso en el que la documentación fuese inconsistente. Para una adaptabilidad óptima, los componentes deberían ser autocontenidos.

En sistemas orientados a objetos, el problema con la herencia es que a medida que se hacen cambios, se vuelve más y más compleja (la adaptación se suele efectuar no modificando un componente, sino creando uno nuevo que herede los atributos y operaciones del original). Los componentes se vuelven más difíciles de entender y la funcionalidad aparece replicada. De este modo, la red de herencia debe ser periódicamente revisada y reestructurada para reducir su complejidad y la duplicación funcional. Por supuesto, esto añade costes adicionales.

Entendibilidad

El cambiar un componente de un diseño implica que la persona responsable de hacer el cambio debe entender el modo de operar de dicho componente. La entendibilidad está relacionada con diversas características de los componentes.

- (1) Cohesión. ¿Puede entenderse el componente sin ayuda de otros componentes?
- (2) Nombres. ¿Son significativos los nombres usados en el componente? Los nombres significativos son aquellos que reflejan los nombres de las entidades del mundo real que modelizan los componentes.
- (3) Documentación. ¿Está el componente documentado de tal modo que la relación entre las entidades del mundo real y el componente están claras? ¿Se ha documentado la razón de ser de dichas relaciones?
- (4) Complejidad. ¿Cómo son de complejos los algoritmos usados para implementar el componente? Una complejidad alta implica numerosas relaciones entre distintas partes del diseño, y una estructura lógica compleja. Los componentes complejos son difíciles de entender, ésta es la razón por la que el diseñador debería concebir componentes lo más sencillo posibles.

Atributos Externos del Software

Los atributos externos del software son aquellos observables a través de su ejecución. Entre ellos encontramos rendimiento, la seguridad, la disponibilidad, la funcionalidad y la usabilidad.

Rendimiento

El rendimiento se refiere a la capacidad de respuesta del sistema. Esto se traduce en el tiempo necesario para responder a eventos (estímulos) o en el número de eventos procesados durante un determinado intervalo de tiempo.


Seguridad

La seguridad es una medida de la eficacia del sistema a la hora de resistir contra los intentos de uso no autorizados mientras se sigue proporcionando servicio a los usuarios legítimos.

Disponibilidad

La disponibilidad mide la proporción de tiempo que el sistema está funcionando. Se mide mediante la cantidad de tiempo entre fallos así como por la rapidez del sistema a la hora de reanudar el servicio en caso de fallo.

Muy estrechamente relacionada con la disponibilidad tenemos a la fiabilidad, que es la habilidad del sistema de mantenerse operativo en el tiempo. La fiabilidad se suele medir usando el tiempo medio entre fallos. Estos dos atributos de calidad están fuertemente ligados al diseño. Las técnicas de diseño incluyen instalar componentes redundantes para recuperarse en caso de fallo, especial atención a la notificación y manejo de errores (que implica restringir los patrones de interacción entre los componentes), y componentes especiales.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

Funcionalidad

La funcionalidad es la habilidad del sistema para hacer el trabajo para el cual se construyó. Desarrollar una tarea requiere que muchos o la mayoría de los componentes del sistema se coordinen para completar un trabajo.

La funcionalidad es ortogonal a la estructura, lo que significa que no es arquitectónica por naturaleza. Se pueden concebir distintas estructuras para implementar una funcionalidad.

El diseño del software restringe la asignación de funcionalidades a estructuras cuando otros atributos de calidad son importantes. Por ejemplo, los sistemas están frecuentemente divididos de tal modo que varias personas pueden cooperar para construirlos. El interés de la funcionalidad reside en cómo interactúa y a la vez restringe, la consecución de estos otros atributos.

Usabilidad

La usabilidad es un término complejo, que normalmente se suele dividir para poder medirlo. Se puede dividir en las siguientes áreas:

- (1) Aprendizaje. ¿Cómo de rápido y fácil es para un usuario aprender a usar la interfaz del sistema?
- (2) Eficiencia. ¿Responde el sistema con una velocidad apropiada a las peticiones del usuario?
- (3) Memorabilidad. ¿Puede recordar el usuario cómo efectuar operaciones con el sistema entre usos del mismo?
- (4) Evitar errores. ¿Anticipa y previene el sistema errores comunes de usuarios?
- (5) Manejo de errores. ¿Ayuda el sistema al usuario a recuperarse de errores?
- (6) Satisfacción. ¿Facilita el sistema el trabajo del usuario?

El hacer la interfaz de usuario de un sistema limpia y fácil de usar es principalmente un asunto de conseguir los detalles correctos de una interacción con el usuario.

Atributos Internos del Software

Modificabilidad

La modificabilidad, en todas sus formas, es un atributo de calidad cercano al diseño del sistema, ya que la habilidad de hacer cambios rápida y rentablemente se deriva directamente de su estructura. La modificabilidad es función de la localidad de cualquier cambio puesto que hacer un cambio extensible al sistema es más costoso que hacer un cambio a un sólo componente y dejar el resto de las cosas igual.

Las modificaciones a un sistema a menudo varían desde cambios en las necesidades de negocio o de la organización. Pueden ser clasificadas de este modo:

- (1) Extender o cambiar capacidades. Añadir nuevas funcionalidades, mejorar las existentes o reparar fallos. La habilidad para adquirir nuevas características se llama extensibilidad. Añadir nuevas capacidades es importante para permanecer competitivos contra otros productos en el mismo mercado.
- (2) Borrar capacidades no deseadas. Para simplificar la funcionalidad de una aplicación existente, quizá entregar una versión de un producto menos capaz (y menos cara).
- (3) Adaptar a nuevos entornos operativos. Por ejemplo, procesadores, dispositivos de entrada/salida, y dispositivos lógicos.
- (4) Reestructuración. Por ejemplo, racionalizando sistemas de servicios, modularizando, optimizando, o creando componentes reutilizables que pueden servir a la hora de crear nuevos sistemas.

La modificabilidad se denomina comúnmente mantenibilidad.

Portabilidad

La portabilidad es la habilidad del sistema para ser ejecutado en diferentes entornos de computación. Estos entornos pueden ser hardware, software o una combinación de los dos. Un sistema es portable hasta el punto de que todas las asunciones acerca de un entorno computacional particular se confinan a un componente (o si acaso, a un número pequeño de componente fácilmente modificables).

Reusabilidad

La reusabilidad se toma usualmente para reflejar que el diseño de un sistema es tal que la estructura del sistema o de alguno de sus componentes puede ser reutilizada en futuras aplicaciones. El diseño para reutilizar significa que el sistema ha sido estructurado de tal modo que sus componentes pueden tomarse de productos anteriormente construidos, en cuyo caso es un sinónimo de integrabilidad. En algunos casos, la reutilización puede ser concebida como otro caso especial de modificabilidad.

La reusabilidad está relacionada con el diseño del software en tanto en cuanto los componentes del diseño son las unidades de reutilización, y el grado de reusabilidad de un componente depende de cómo esté de acoplado con otros.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

Integrabilidad

Es la habilidad de hacer funcionar aquellos componentes de un sistema que han sido desarrollados de modo separado. Esto por otro lado depende de la complejidad externa de los componentes, sus mecanismos de interacción y protocolos, y el grado hasta el cual las responsabilidades han sido particionadas de modo limpio. La integrabilidad también depende del grado en que se hayan especificado los componentes.

Integrar un componente depende no sólo de los mecanismos de interacción usados, sino también de la funcionalidad asignada al componente al ser integrado y como esa funcionalidad está relacionada con la funcionalidad del entorno del nuevo componente.

Un tipo especial de integrabilidad es la interoperatividad. La integrabilidad mide la habilidad de las partes de un sistema a la hora de trabajar juntas; la interoperatividad mide la habilidad de un grupo de partes (constituyendo un sistema) para trabajar con otro sistema.

Testeabilidad

La testeabilidad del software se refiere a la facilidad con la cual el software puede ser construido para mostrar sus fallos a lo largo de las pruebas. En particular, la testeabilidad se refiere a la probabilidad de que, asumiendo que el software tiene al menos un fallo, el software fallará en su próxima ejecución de prueba. La testeabilidad está relacionada con los conceptos de observabilidad y controlabilidad. Para que un sistema sea testeable, debe ser posible controlar cada estado interno del componente y entradas y entonces observar sus salidas.

La testeabilidad de un sistema se relaciona con otros asuntos estructurales o de diseño: su nivel de documentación, separación de intereses y el grado hasta el cual el sistema oculta la información. El desarrollo incremental también beneficia la testeabilidad del mismo modo que mejora la integrabilidad.

Técnicas de Diseño

Diseño Funcional

En esta estrategia de diseño, el sistema se diseña desde un punto de vista funcional, comenzando con una vista a alto nivel que se va refinando progresivamente hasta conseguir un diseño detallado. El estado del sistema es único en cada momento, ya que está centralizado y es por tanto compartido por todas las funciones que operan en ese estado.

El ejemplo por excelencia de esta estrategia es el método estructurado de Yourdon y Constantine. Los métodos de programación estructurada de Jackson o el método Warnier-Orr también son técnicas de descomposición funcional pero a diferencia de la anterior, en éstas la estructura de los datos se usa para determinar la estructura funcional necesaria para procesar dichos datos.

Diseño Orientado a Objetos

En esta técnica, el sistema se ve como una colección de objetos más que como funciones.

El estado del sistema está descentralizado y cada objeto gestiona su propia información de estado. Los objetos tienen un conjunto de atributos que definen su estado y operaciones que actúan sobre dichos atributos. Los objetos suelen ser miembros de una clase cuya definición especifica atributos y operaciones para esa clase. Pueden ser heredados de una o más superclases de tal modo que la definición de una clase necesita solamente definir las diferencias entre dicha clase y su superclase. Conceptualmente, los objetos se comunican usando paso de mensajes, aunque en la práctica suelen hacerlo mediante llamadas a procedimientos.

El diseño orientado a objetos está basado en la idea de Parnas de ocultación de la información, y ha sido descrito por autores como Meyer, Booch o Rumbaugh.

Método de Desarrollo de Sistemas de Jackson

El método de diseño de sistemas de Jackson cae en algún lugar entre los orientados a funciones y el diseño orientado a objetos. Es una evolución del trabajo realizado por Jackson en el método de programación estructurada. Similar de algún modo al enfoque de Warnier, este método se centra en los modelos del dominio de información del *mundo real*. Para realizar el DSJ, se siguen los siguientes pasos:

- (1) *Entidades y acciones*. Mediante un enfoque similar a la técnica seguida por el método de orientación a objetos, se identifican las entidades (personas, objetos u organizaciones que un sistema necesita para producir o utilizar información) y las acciones del sistema (los sucesos que ocurren en el mundo real que afectan a las entidades).
- (2) *Estructura de entidades*. Las acciones que afectan a cada entidad se ordenan cronológicamente y se representan con una notación tipo árbol.
- (3) *Modelizado inicial*. Las entidades y acciones se representan como un modelo de proceso, se definen las conexiones entre el modelo y el mundo real.
- (4) *Planificación del sistema*. Se evalúan y se especifican las características de planificación de procesos.
- (5) *Implementación*. El hardware y el software se especifican como un diseño.

Diseño de los datos

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

El diseño de los datos es una actividad importante a la hora de realizar un diseño. El hecho de realizar un modelo de los datos, depurarlo y llegar a estructuras independientes, no exige pensar en ningún momento en la utilización de una base de datos. Por ello, se debe llegar a la forma más depurada de este modelo y luego que sea el administrador del sistema el que se encargue de decidir la forma en que se implementará dicho modelo.

Una de las técnicas más usadas a la hora de construir el modelo de datos de un sistema es el modelo entidad-relación de Chen. Es un modelo n-ario, lo que significa que las relaciones pueden asociar una, dos o más entidades. Se puede hablar de relaciones unitarias, binarias o ternarias.

Existen, además, dos tipos de entidades representables, las regulares y las débiles.

Al modelo construido según la técnica de Chen se le aplica una serie de refinamientos sucesivos, mediante los cuales se obtienen un conjunto de tablas, que serán posteriormente normalizadas. A la hora de construir el modelo, se identifican las siguientes fases:

- (1) *Construcción del Modelo.* En este paso, se identifican las entidades del sistema, se determinan los identificadores de las entidades, se establecen las relaciones entre las entidades, describiendo la cardinalidad de las mismas, se dibuja el modelo de datos y se identifican y describen los atributos de cada entidad. Por último se verifica el modelo.
- (2) *Normalización del modelo.* Donde se reducen las inconsistencias y las redundancias de los datos. Facilita el mantenimiento de los datos, evita anomalías en operaciones de manipulación de datos y reduce el impacto de los cambios en los datos.
- (3) *Optimización del modelo.* Para asegurar que se cumplan los requisitos de rendimiento del sistema. Se utilizarán diversas técnicas.

Diseño Orientado a Eventos

Existen ciertos tipos de aplicaciones que se caracterizan por depender fuertemente de la interfaz de usuario. Este método de diseño conocido como técnica de guiones se basa en la especificación de las interfaces de usuario; tiene en cuenta las necesidades del usuario a la hora de utilizar el sistema y la secuencia en que se suceden los distintos usos del sistema.

La descripción de las interfaces de usuario se obtiene partiendo de una serie de sesiones con el usuario.

Los pasos que se siguen son:

- (1) Identificación de roles.
- (2) Creación de una maqueta inicial.
- (3) Comprobación de guiones.
- (4) Sesiones con el usuario. Refinamiento de guiones y escenarios.
- (5) Descripción de eventos y procedimientos asociados.

Evaluación del Diseño de Software

Las razones por las cuales se debe evaluar un diseño software son numerosas, pero quizá la más importante de ellas sea la conveniencia de evaluar el software lo antes posible en el ciclo de vida, para así reducir los costes de corrección de los defectos encontrados. Aunque también hay otras razones, como por ejemplo la consideración de posibles alternativas de diseño antes de tomar una decisión final.

A la hora de evaluar un diseño, por supuesto hará falta una descripción o una especificación del mismo, pero, ¿qué tipo de descripción o especificación? La respuesta es, aquella que necesite la evaluación. Si la evaluación se centra en el rendimiento o en el paralelismo, se necesitará una descripción de las tareas que realiza y probablemente la estructura de las comunicaciones. Si por el contrario nos centramos en la modificabilidad, se necesitará la descomposición en módulos.


Vamos a presentar dos técnicas de evaluación del diseño del software para determinar si se han cumplido las metas de calidad determinadas por el cliente/usuario: el método de análisis de la arquitectura del software y las revisiones de diseño. La primera de ellas sirve tanto para decir qué alternativa de diseño de entre varias es la mejor, como para determinar si un diseño cumple los objetivos propuestos. La segunda de ellas, servirá únicamente para el segundo de los objetivos.

Método de Análisis de la Arquitectura del Software

Según Kazman et al. (2001), el Método de Análisis de Arquitecturas de Software (Software Architecture Analysis Method, SAAM) es el primero que fue ampliamente promulgado y documentado. El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integridad.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la arquitectura a ser evaluada. De acuerdo con Kazman, las salidas de la evaluación del método SAAM son las siguientes:

- Una proyección sobre la arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones.

La siguiente tabla presenta los pasos que contempla el método de evaluación SAAM, con una breve descripción.

Desarrollo de escenarios	Un escenario es una breve descripción de usos del sistema. Pueden incluir cambios a los que puede estar expuesto el sistema en el futuro.
Descripción de la arquitectura	La arquitectura debe ser descrita haciendo uso de alguna notación arquitectónica que sea común a todas las partes involucradas en el análisis. Deben incluirse los componentes de datos y conexiones relevantes, así como la descripción del comportamiento general del sistema.
Clasificación y asignación de prioridad de los escenarios	La clasificación de escenarios puede hacerse en: directos e indirectos. Un escenario directo es el que puede satisfacerse sin la necesidad de modificaciones en la arquitectura. Un escenario indirecto es aquel que requiere modificaciones en la arquitectura para poder satisfacerse.
Evaluación de escenarios indirectos	Para cada escenario indirecto, se listan los cambios necesarios sobre la arquitectura, y se calcula su costo.
Evaluación de la interacción entre escenarios	Cuando dos o más escenarios indirectos proponen cambios sobre un mismo componente, se dice que interactúan sobre ese componente. Es necesario evaluar este hecho, puesto que la interacción de componentes semánticamente no relacionados revela que los componentes de la arquitectura efectúan funciones semánticamente distintas.
Creación de la evaluación global	Debe asignársele un peso a cada escenario, en términos de su importancia relativa al éxito del sistema. Esta asignación de peso suele hacerse con base en las metas del negocio que cada escenario soporta.

Revisiones de Diseño

Existen distintas técnicas para llevar a cabo una revisión de diseño. Cada una de ellas tiene distintos costes asociados y puede usarse para obtener distintos tipos de información.

Hay dos categorías básicas: aquellas que generan preguntas cualitativas para hacer a un diseño y las que sugieren medidas cuantitativas que aplicar a un diseño. Las técnicas de preguntas se pueden aplicar para evaluar cualquier atributo del diseño, no obstante, no proporcionan medios para contestar a estas preguntas. Las técnicas de medida, por otro lado, se usan para contestar preguntas específicas. En este sentido, sirven para atributos de calidad específicos.


A continuación se describirán tres técnicas de preguntas: escenarios, cuestionarios y listas de comprobación:

- (1) *Técnicas basadas en escenarios.* Describen una interacción específica entre el cliente/usuario y el sistema a desarrollar.
- (2) *Técnicas basadas en cuestionarios.* Un cuestionario es una lista de preguntas genéricas que se pueden aplicar a todos los diseños. Su utilidad está relacionada con la facilidad con la cual el dominio de interés se puede caracterizar, ya que en este tipo de revisiones, el equipo de revisión explorará cada pregunta al nivel de detalle necesario para contestarla.
- (3) *Técnicas basadas en listas de comprobación.* Una lista de comprobación es un conjunto detallado de preguntas que se desarrolla después de conseguir mucha experiencia evaluando un determinado tipo de sistemas. Éstas tienden a centrarse mucho más en atributos particulares del sistema.

La relación entre estas técnicas es que la experiencia revisando el mismo tipo de sistema puede resultar bien en una lista de comprobación (si es específica del dominio) o en un cuestionario (cosas más generales). Las listas de comprobación y los cuestionarios reflejan prácticas de revisión más maduras. Además, los escenarios se construyen en base al sistema a revisar y las otras existen de sistemas a otros.

Como técnicas de medida tenemos:

- (1) *Métricas.* Son interpretaciones cuantitativas colocadas en medidas observables en el diseño. Con estas técnicas la revisión necesita centrarse no sólo en los resultados de la aplicación de la métrica sino también en las asunciones bajo las cuales la técnica se usó.
- (2) *Simulaciones, prototipos y experimentos.* La construcción de un prototipo o una simulación del sistema puede ayudar a crear y clarificar un diseño. También se pueden usar para contestar una pregunta que surge por una técnica de preguntas.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

ARQUITECTURA DE SOFTWARE

Cada vez se entiende menos la existencia de aplicaciones monolíticas ejecutadas en un único ordenador y sin comunicación externa. Las grandes aplicaciones corriendo como procesos muy pesados en potentes ordenadores es ya un capítulo de la historia. La interconexión y la distribución son las características emergentes del nuevo software y constituyen elementos estratégicos para la supervivencia de cualquier empresa. En un panorama en el que aumenta progresivamente el número de aplicaciones distribuidas y el tamaño y complejidad de las mismas, es preciso dotar a los ingenieros software de herramientas capaces de abordar la construcción y mantenimiento de dichas aplicaciones. Además, el software resultante debería ser un "software de calidad"; es decir, debería ser correcto, robusto, extensible, reutilizable, compatible, eficiente, portable, verificable y fácil de usar. Una de dichas herramientas es el establecimiento de un estilo de arquitectura software para el sistema en cuestión.

La expansión exponencial de Internet, y especialmente del World-Wide Web, abrió nuevas oportunidades de desarrollo software. A diferencia del desarrollo software tradicional en el que el producto lo controla una única organización, ahora nos enfrentamos a aplicaciones compuestas por una coalición de recursos autónomos y distribuidos. Estos recursos se desarrollan y mantienen de forma independiente, pueden ser transitorios (aparecer/desaparecer la red de forma dinámica) y obedecen a políticas de quién los desarrolló aunque se ponen a disposición de uso externo. A falta de una autoridad central para el control de dichos recursos, se imponen un sistema de coalición (federaciones). A continuación nombraremos algunos requisitos, métodos de desarrollo y tecnologías que acompañan al cambio global en la computación:

- (1) La importancia creciente del uso de componentes prefabricados que nos conducirá a menores tiempos de desarrollo y a una más fácil personalización de dichos componentes (componentware).
- (2) La aceptación creciente de los métodos de diseño orientados a objetos para abordar la actual complejidad de los sistemas software.
- (3) Middleware como CORBA (Common Object Request Broker Architecture), Java RMI (Remote invocation method) o MOM (Message-Oriented Middleware) que proporcionan infraestructuras para soportar la cooperación de componentes heterogéneos (imprescindible en el desarrollo de aplicaciones a gran escala).

También es importante subrayar que estamos asistiendo a un aumento de la demanda tecnológica para sistemas distribuidos, aplicaciones Web, técnicas de integración de las nuevas tecnologías con sistemas ya existentes (legacy systems), incremento en la complejidad de los sistemas de información e integración de tecnologías dispares (por ejemplo, sistemas orientados a objetos deben convivir con sistemas de gestión de bases de datos relacionales). Estos y otros desafíos de esta nueva generación de aplicaciones, implican crear nuevos procesos de desarrollo software o adaptar los ya existentes. El diseño de la arquitectura de la aplicación, como una de las actividades del proceso de desarrollo software, no es ajeno a estas nuevas necesidades. Es por ello, que si queremos ser capaces de afrontar con éxito la generación actual (y futura) de aplicaciones, que ya desbordan las capacidades de las típicas arquitecturas cliente-servidor sobre LAN, es imprescindible el desarrollo de nuevos estilos arquitectónicos y nuevos esquemas de interacción que permitan la definición de arquitecturas escalables, interoperables, reutilizables, dinámicas, evolutivas y reflexivas.


Conceptos de Sistemas Distribuidos

Un sistema distribuido está formado por una colección de ordenadores autónomos conectados por una red y equipado con un software que permite a dichos ordenadores coordinar sus actividades y compartir recursos (hardware, software y datos).

Conceptos Básicos

Las características claves de todo sistema distribuido son las siguientes:

- (1) Recursos compartidos, entendiendo como recursos desde componentes hardware como discos o impresoras a entidades software tales como ficheros y bases de datos.
- (2) Extensibilidad, entendida esta como la característica que indica si el sistema puede ser extendido en varias direcciones (extensiones hardware y/o software). Los sistemas que poseen esta característica proporcionan un mecanismo uniforme de comunicación entre procesos y hacen públicas las interfaces de acceso a los recursos compartidos.
- (3) Concurrencia, que se origina de forma natural en los sistemas distribuidos debido a las actividades independientes de los diferentes usuarios, la independencia de los recursos y la localización de procesos servidores en diferentes ordenadores.
- (4) Escalabilidad, que se refiere a que el software no debería precisar de cambios cuando se incrementa la escala del sistema.
- (5) Tolerancia a fallos, entendiendo que un sistema es tolerante a fallos si puede detectar un fallo y entonces o bien detenerse elegantemente o bien enmascarar el fallo de tal forma que el usuario no lo perciba. Esta característica se puede conseguir mediante la redundancia hardware y/o software recuperable (programas diseñados para recuperarse de fallos). Relacionada con la tolerancia a fallos, está la característica de la disponibilidad entendida ésta como una medida de la proporción de tiempo que está disponible un sistema para usarlo.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

(6) Transparencia, definida como el ocultamiento al usuario y a los programadores de aplicaciones de la separación de componentes en un sistema distribuido, de tal forma que el sistema se percibe como un conjunto más que como una colección de componentes independientes. La International Standard Organization (ISO) identifica hasta 8 formas de transparencia:

- Transparencia de acceso, que permite que los objetos locales y remotos se accedan utilizando idénticas operaciones.
- Transparencia de localización, que permite acceder a los objetos sin conocer su localización física.
- Transparencia de concurrencia, que permite a varios procesos operar concurrentemente sobre objetos compartidos sin interferir entre ellos.
- Transparencia de replicación, que permite utilizar múltiples instancias de objetos para incrementar la fiabilidad y el rendimiento sin que dichas réplicas sean conocidas por los programas de aplicación o los usuarios.
- Transparencia de fallos, que permite a usuarios o programas de aplicación completar sus tareas a pesar de fallos en componentes software o hardware.
- Transparencia de migración, que permite mover objetos de un nodo a otro sin que ello afecte a las operaciones de los usuarios o de los programas de aplicación.
- Transparencia de rendimiento, que permite reconfigurar el sistema para mejorar su rendimiento ante la presencia de variaciones en la carga.
- Transparencia de escala, que permite acomodar un crecimiento de carga (más componentes y más peticiones) sin cambiar la estructura del sistema o los algoritmos de la aplicación.


La suma de la transparencia de acceso y de localización es lo que se denomina transparencia de red.

Otros autores añaden una característica más: Acceso y Seguridad. Según estos autores, la seguridad de un sistema distribuido incluye tres aspectos:

- Control de acceso. Se deben comprobar los permisos del cliente para todas y cada una de las operaciones que se pueden realizar sobre todos y cada uno de los recursos.
- Autenticación. Por cada petición de servicio se debe autenticar la identidad del cliente que lo invoca.
- Auditoría. Se debería poder registrar cada operación realizada.

Las siete características indicadas anteriormente podrían considerarse como requisitos genéricos para todo sistema distribuido y cuyo soporte tendrá que venir dado por la arquitectura del sistema obtenida durante la fase de diseño. Dicho diseño persigue conseguir alto rendimiento, fiabilidad, escalabilidad, consistencia y seguridad. Los problemas que tiene que abordar un diseñador en la construcción de un sistema distribuido y que devienen específicamente de la propia naturaleza del sistema distribuido son los siguientes:

- Nombrado: se debe diseñar un sistema de nombrado que sea fácilmente escalable y con un esquema de traducción eficiente para cumplir con los objetivos de escalabilidad y rendimiento.
- Comunicación: el rendimiento y la fiabilidad de las técnicas de comunicación empleadas para la implementación del sistema distribuido, son críticos para el rendimiento global del sistema. Si el mecanismo de comunicación es el paso de mensajes, habrá que determinar si es en forma síncrona o asíncrona. Si el patrón de comunicación es cliente-servidor habrá que determinar si se utilizará RPC o no; si utilizamos un patrón para comunicar grupos de procesos cooperantes habrá que determinar si existe la facilidad multicast o debemos implementarla nosotros.
- Estructura software: la extensibilidad se consigue diseñando componentes con interfaces bien definidas. Hay que estructurar el sistema de tal forma que puedan introducirse nuevos servicios que se integren totalmente con los existentes y sin duplicar elementos de servicio que ya existen.
- Asignación de carga: aquí el problema de diseño es cómo distribuir el procesamiento, las comunicaciones y los recursos para optimizar el rendimiento incluso cuando varía la carga en los nodos. Habrá que elegir algún modelo como puede ser el modelo de pool de procesadores, el modelo de uso de ordenadores ociosos o el modelo de multiprocesadores con memoria compartida.
- Mantenimiento de la consistencia: el mantenimiento de la consistencia a un costo razonable (en cuanto a su impacto en el rendimiento) es quizás el problema más difícil del diseño de un sistema distribuido. Distinguiremos varios tipos de consistencia:
 - Consistencia en la actualización: el problema se origina cuando varios procesos acceden y actualizan datos concurrentemente.
 - Consistencia en la replicación: las réplicas deben ser consistentes entre sí.
 - Consistencia en la caché: el problema se origina cuando algunos datos están en la caché de un cliente y son actualizados por otro.
 - Consistencia en los fallos: Si las operaciones en algún nodo dependen del funcionamiento de otro nodo y éste último falla, en algún momento se producirá un fallo en el primero. Se necesitan procedimientos de rollback que dejen los datos en un estado consistente.
 - Consistencia en los relojes: relacionado con el problema de la sincronización de relojes en cada ordenador.
 - Consistencia en la interfaz de usuario: en un sistema distribuido, si una entrada de usuario tiene que tratarse remotamente, existe un lapsus de tiempo durante el cual el aspecto de la pantalla (la interfaz) es inconsistente con el modelo de aplicación.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

Además, no existen soluciones enlatadas para construir grandes sistemas orientados a objetos distribuidos y los problemas típicos de la distribución tales como el nombrado, la fiabilidad, la disponibilidad, la replicación, la tolerancia a fallos, la integridad transaccional, la persistencia, la seguridad, la movilidad de objetos, la heterogeneidad, la escalabilidad y el rendimiento deben ser replanteados de una forma integrada para cada aplicación debido a los diferentes compromisos que deben adquirirse en cada una de ellas.

Por otro lado, los sistemas distribuidos presentan algunos inconvenientes:

- (1) Complejidad. Los sistemas distribuidos son más complejos que los sistemas centralizados. Esto hace más difícil comprender sus propiedades emergentes y verificar estos sistemas. Por ejemplo, en lugar de medir el rendimiento del sistema por la velocidad del procesador, ahora hay que medirlo en función del ancho de banda de la red y en función de la velocidad de los diferentes procesadores involucrados en el cálculo. Mover recursos de un nodo a otro puede afectar de manera dramática al rendimiento.
- (2) Seguridad. Al sistema se puede acceder desde diferentes ordenadores y el tráfico de la red está sujeto a escuchas piratas.
- (3) Gestión del sistema. Los ordenadores de un sistema pueden ser de tipos diferentes y pueden ejecutar sistemas operativos diferentes. Los fallos en una máquina se pueden propagar a otras máquinas con consecuencias imprevisibles.
Esto significa que hace falta más esfuerzo para gestionar y mantener el sistema en funcionamiento.
- (4) Impredecibilidad. Los sistemas distribuidos son impredecibles en sus respuestas. La respuesta depende de la carga global del sistema, su organización y la carga de la red. Como cualquiera de estos factores puede cambiar en un corto espacio de tiempo, el tiempo de respuesta a una petición de un usuario puede variar enormemente de una petición a otra.

Conceptos de Arquitectura de Software

La arquitectura software se ha convertido en un área de intensa investigación dentro de la comunidad de la ingeniería del software. La arquitectura trata con un problema esencialmente difícil: disminuir la complejidad de una aplicación mediante la abstracción y la reutilización. Debido a la reciente aparición de esta disciplina, existen aún diferentes criterios respecto a lo que es y no es una arquitectura software. De hecho, algunos autores prefieren empezar definiendo lo que no es una arquitectura. Por ejemplo algunas concepciones erróneas de arquitectura software son: (1) el equívoco entre arquitectura software y los lenguajes de descripción de arquitecturas (ADLs), (2) la equiparación de la arquitectura con el proceso de diseño y (3) la arquitectura como un vehículo para los primeros estadios del análisis. Por otra parte, algunos autores exponen sus definiciones de arquitectura software. Por ejemplo: "La arquitectura software de un programa o sistema informático es la estructura o estructuras del sistema, que comprende componentes software, las propiedades externamente visibles de estos componentes y las relaciones entre ellos". En Shaw y Garlan (1996) podemos leer: "Brevemente, una arquitectura software involucra la descripción de los elementos a partir de los cuales se construyen los sistemas, las interacciones entre dichos elementos, patrones que guían su composición y restricciones sobre esos patrones. En general un sistema particular se define en términos de una colección de componentes e interacciones entre esos componentes".

Conceptos Básicos

En general, podemos decir que toda arquitectura software define el conjunto de componentes que la integran y el modo en que dichos componentes interactúan unos con otros. Para describir dichos componentes y su interacción es necesario disponer de un conjunto de herramientas específicas. Serán necesarios, lenguajes de definición de arquitecturas, métodos de diseño, herramientas de ayuda al diseño, métodos formales específicos, etc.

Existen tres conceptos básicos en toda arquitectura software: el estilo arquitectónico, que es la descripción de los tipos de componentes y un patrón de su control de ejecución y/o transferencia de datos; el modelo de referencia, que es una división de las funcionalidades junto con los flujos de datos entre las piezas; y una arquitectura de referencia, que es el modelo de referencia planeado para los componentes software (que implementarán las funcionalidades definidas en el modelo de referencia) y los flujos de datos entre componentes. Por ejemplo, para el caso de un compilador:

- (1) El modelo de referencia contiene el conocimiento que tenemos sobre la división de las funcionalidades del compilador (analizador léxico, sintáctico, generador de código, optimizador, etc.), así como el flujo de datos entre esas partes (el analizador léxico proporciona tokens al analizador sintáctico).
- (2) El estilo arquitectónico podría ser pipe&filters.
- (3) La arquitectura de referencia obtenida asignaría la funcionalidad descrita en el modelo de referencia a cada uno de los filters y describiría el flujo de información a través de los pipes.

El estilo arquitectónico junto con el modelo de referencia determinan la arquitectura de referencia que es, a su vez, un paso previo a la definición de una arquitectura software concreta. En el ejemplo anterior del compilador, con otro estilo arquitectónico (por ejemplo, memoria compartida) obtendríamos otra arquitectura de referencia (manteniendo el mismo modelo de referencia).

Existen diferentes estilos arquitectónicos que serán más o menos apropiados según el sistema concreto que se desee construir. Se han utilizado los siguientes discriminantes para proponer una taxonomía de arquitecturas:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

- (1) Qué géneros de componentes y conectores se utilizan.
- (2) Cómo se transfiere el control entre componentes.
- (3) Cómo fluyen los datos a través del sistema.
- (4) Cómo interactúan los datos y el control.
- (5) Qué tipo de razonamiento es compatible con el estilo arquitectónico.

Estilos Arquitectónicos

Un estilo arquitectónico viene definido por los siguientes elementos:

- (1) Un conjunto de tipos de componentes (por ejemplo, repositorio de datos, proceso, procedimiento) que ejecutan alguna función en tiempo de ejecución.
- (2) Una estructura topológica que indica las interrelaciones de estos componentes en tiempo de ejecución.
- (3) Un conjunto de restricciones semánticas (por ejemplo, un repositorio de datos no permite que se cambien los datos almacenados en él).
- (4) Un conjunto de conectores (por ejemplo, llamada a subrutina, flujo de datos, sockets) que median la comunicación, coordinación o cooperación entre componentes.

Entre los estilos arquitectónicos más conocidos encontramos:

Modelo de repositorio

La mayoría de los sistemas que usan gran cantidades de datos se organizan alrededor de una base de datos compartida o repositorio. Por lo tanto, este modelo es adecuado para aplicaciones en las que los datos son generados por un subsistema y son usados por otro. Entre sus ventajas encontramos:

- (1) Es una forma eficiente de compartir grandes cantidades de datos. No hay necesidad de transmitir datos explícitamente de un subsistema a otro.
- (2) Los subsistemas que producen datos no necesitan conocer cómo se utilizan sus datos por otros subsistemas.
- (3) Las actividades tales como copias de seguridad, protección, control de acceso y recuperación de errores están centralizadas. Son las responsables de gestionar el repositorio. Las herramientas pueden centrarse en su función principal en lugar de estar relacionadas con estas cuestiones.
- (4) El modelo de compartición es visible a través del esquema del repositorio. Las nuevas herramientas se integran de forma directa puesto que estas son compatibles con el modelo de datos acordado.

Modelo cliente-servidor

Este modelo de sistema se organiza como un conjunto de servicios y servidores asociados, más unos clientes que acceden y usan los servicios. Los principales componentes de este modelo son:

- (1) Un conjunto de servidores que ofrecen servicios a otros subsistemas.
- (2) Un conjunto de clientes que llaman a los servicios ofrecidos por los servidores. Estos son normalmente subsistemas en sí mismos. Puede haber varias instancias de un programa cliente ejecutándose concurrentemente.
- (3) Una red que permite a los clientes acceder a estos servicios.


Los clientes pueden conocer los nombres de los servidores disponibles y los servicios que estos proporcionan. Sin embargo, los servidores no necesitan conocer la identidad de los clientes o cuántos clientes tienen. Los clientes acceden a los servicios proporcionados por un servidor a través de llamadas a procedimientos remotos usando un protocolo de petición-respuesta.

La ventaja más importante del modelo es que es una arquitectura distribuida. Se puede hacer un uso efectivo de los sistemas en red con muchos procesadores distribuidos. Es fácil añadir un nuevo servidor e integrarlo con el resto del sistema o actualizar los servidores de forma transparente sin afectar al resto del sistema.

Modelo de capas

El modelo de capas de una arquitectura organiza el sistema en capas, cada una de las cuales proporciona un conjunto de servicios. Cada capa puede pensarse como una máquina abstracta cuyo lenguaje máquina se define por los servicios proporcionados por la capa. Este "lenguaje" se usa para implementar el siguiente nivel de la máquina abstracta.

La aproximación por capas soporta el desarrollo incremental de sistemas. A medida que se desarrolla una capa, algunos de los servicios proporcionados por esa capa pueden estar disponibles para los usuarios. Esta arquitectura soporta bien los cambios y es portable. En la medida en la que su interfaz permanezca sin cambios, una capa puede reemplazarse por otra capa equivalente. Además, cuando las interfaces de la capa cambian o se añaden nuevas facilidades a una capa, solamente se ve afectada la capa adyacente. Debido a que los sistemas por capas localizan las dependencias de la máquina en las capas más internas, es mucho más fácil proporcionar implementaciones

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.2 VIGENCIA: 19-09-2007
	APUNTE DE TEMAS DE REVISIÓN	

multiplataforma de las aplicaciones de un sistema. Únicamente las capas más internas dependientes de la máquina necesitan ser reimplementadas para tener en cuenta las facilidades de un sistema operativo o base de datos diferente.

Modelo orientado a objetos

Un modelo arquitectónico orientado a objetos estructura el sistema en un conjunto de objetos débilmente acoplados y con interfaces bien definidas. Los objetos realizan llamadas a los servicios ofrecidos por otros objetos.

Las ventajas de la orientación a objetos son bien conocidas, debido a que los objetos están débilmente acoplados, la implementación de los objetos pueden modificarse sin afectar a otros objetos. Los objetos son a menudo representaciones de entidades del mundo real por lo que la estructura del sistema es fácilmente comprensible. Debido a que las entidades del mundo real se usan en sistemas diferentes, los objetos pueden reutilizarse.

Modelo orientado a flujos de funciones

En una descomposición orientada a flujos de funciones o modelo de flujos de datos, las transacciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de una función a otra y se transforman a medida que se mueven a través de la secuencia de funciones. Cada paso de procesamiento se implementa como una transformación. Los datos de entrada fluyen a través de estas transformaciones hasta que se convierten en datos de salida. Las transformaciones se pueden ejecutar secuencialmente o en paralelo. Los datos pueden ser procesados por cada transformación elemento a elemento o en un único lote.

Cuando las transformaciones se representan como procesos separados, algunas veces este modelo se denomina modelo de tubería y filtro. El término *filtro* se usa debido a que una transformación “filtra” los datos que puede procesar desde su flujo de datos de entrada.

Cuando las transformaciones son secuenciales con datos procesados por lotes, este modelo arquitectónico es un modelo secuencial por lotes. Esta arquitectura es común para sistemas de procesamiento de datos, ya que estos generan muchos informes de salida que se derivan a partir de cálculos simples sobre un gran número de registros de entrada.

Las ventajas de esta arquitectura son las siguientes:

- (1) Permite la reutilización de transformaciones.
- (2) Es intuitiva puesto que muchas personas piensan en su trabajo en términos de procesamiento de entradas y salidas.
- (3) Generalmente se puede hacer evolucionar de forma directa el sistema añadiendo nuevas transformaciones.
- (4) Es sencilla de implementar ya sea como un sistema concurrente o como uno secuencial.

Modelo de control centralizado

En un modelo de control centralizado, un subsistema se diseña como el controlador del sistema y tiene la responsabilidad de gestionar la ejecución de otros subsistemas. Los modelos de control centralizado se dividen en dos clases, según que los subsistemas controlados se ejecuten secuencialmente o en paralelo.

- (1) *El modelo de llamada-retorno.* Es el modelo usual de subrutina descendente en donde el control comienza al inicio de una jerarquía de subrutinas y, a través de las llamadas a subrutinas, el control pasa a niveles inferiores en el árbol de la jerarquía. El modelo de subrutinas solamente es aplicable a sistemas secuenciales.
- (2) *El modelo del gestor.* Este es aplicable a sistemas concurrentes. Un componente del sistema se diseña como un gestor del sistema y controla el inicio, parada y coordinación del resto de los procesos del sistema. Un proceso es un subsistema o módulo que puede ejecutarse en paralelo con otros procesos. Una variante de este modelo también puede aplicarse a sistemas secuenciales en los que la rutina de gestión llama a subsistemas particulares dependiendo de los valores de algunas variables de estado.

El modelo de llamada-retorno no es un modelo estructural. La naturaleza rígida y restrictiva de este modelo es tanto una ventaja como un inconveniente. Es una ventaja debido a que es relativamente simple analizar flujos de control y conocer cómo responderá el sistema a cierto tipo de entradas. Es un inconveniente debido a que las excepciones a las operaciones normales son tediosas de manejar.

El modelo del gestor se usa a menudo en sistemas de tiempo real “blandos”, los cuales no tienen restricciones de tiempo muy estrictas. El controlador central gestiona la ejecución de un conjunto de procesos asociados con sensores y actuadores.

El proceso controlador del sistema decide cuándo deberían comenzar o terminar los procesos dependientes de variables de estado del sistema. El controlador por lo general realiza ciclos continuamente, consultando los sensores y otros procesos para detectar eventos o cambios de estado. Por esta razón, este modelo se llama algunas veces modelo de ciclo de eventos.

Modelo dirigido por eventos

Los modelos de control dirigidos por eventos se rigen por eventos generados externamente. El término *evento* en este contexto no solo significa una señal binaria. Puede ser una señal dentro de un rango de valores o una entrada de un

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DE REVISIÓN	VERSIÓN: 1.2 VIGENCIA: 19-09-2007

comando desde un menú. La diferencia entre un evento y una entrada simple es que la aparición del evento está fuera del control del proceso que maneja ese evento.

Podemos analizar dos modelos de control dirigidos por eventos:

- (1) *Modelos de transmisión (broadcast)*. En estos modelos, un evento se transmite a todos los subsistemas. Cualquier subsistema que haya sido programado para manejar ese evento puede responder a él.
- (2) *Modelos dirigidos por interrupciones*. Estos se usan exclusivamente en sistemas de tiempo real, en donde las interrupciones externas son detectadas por un manejador de interrupciones. A continuación, estas se envían a algún otro componente para su procesamiento.

Los modelos de transmisión son efectivos para integrar subsistemas distribuidos en diferentes computadoras de una red. Los modelos dirigidos por interrupciones se usan en sistemas de tiempo real con requerimientos temporales rígidos.

La ventaja de la aproximación de transmisión es que la evolución es relativamente simple. Un nuevo subsistema puede integrarse para manejar clases particulares de eventos registrando sus eventos con el manejador de eventos. Cualquier subsistema puede activar otros subsistemas sin conocer su nombre o localización.

El modelo de control dirigido por interrupciones se usa principalmente en sistemas de tiempo real en los que es necesaria una respuesta inmediata a algún evento. Puede ser combinado con el modelo de gestión centralizado. La ventaja de esta aproximación es que permite implementar respuestas muy rápidas a los eventos.

Modelo de dominio específico

Los modelos arquitectónicos antes enunciados son modelos generales; a estos se pueden sumar los modelos arquitectónicos que son específicos para un dominio particular de aplicación. Sin bien las instancias de estos sistemas difieren en los detalles, la estructura arquitectónica común puede reutilizarse cuando se desarrollan nuevos sistemas. Estos modelos arquitectónicos se denominan *arquitecturas de dominio específico*.

Hay dos tipos de modelos:

- (1) *Modelos genéricos*. Son abstracciones obtenidas a partir de varios sistemas reales. Encapsulan las características principales de estos sistemas.
- (2) *Modelos de referencia*. Son más abstractos y describen una clase más amplia de sistemas. Constituyen un modo de informar a los diseñadores sobre la estructura general de esta clase de sistemas. Los modelos de referencia normalmente se obtienen a partir de un estudio del dominio de la aplicación. Representan una arquitectura ideal que incluye todas las características que los sistemas podrían incorporar.

No hay una distinción rígida entre estos tipos de modelos. Los modelos genéricos también pueden servir como modelos de referencia. Se plantea la diferencia entre ellos, ya que los modelos genéricos pueden reutilizarse directamente en un diseño. Los modelos de referencia se usan normalmente para comunicar conceptos del dominio y comparar o evaluar posibles arquitecturas.

BIBLIOGRAFÍA

[Pressman97] Roger S. Pressman (1997). "Ingeniería del Software: Un enfoque práctico" Mc Graw Hill. España
 [Sommerville05] Ian Sommerville (2005). "Ingeniería del Software" Pearson Addison Wesley. España
 [Vegas04] Sira Vegas Hernández (2004). "Análisis y modelización de las necesidades del usuario" Facultad de Informática – Universidad Politécnica de Madrid. España
 [Vegas04] Sira Vegas Hernández (2004). "Introducción al diseño de software" Facultad de Informática – Universidad Politécnica de Madrid. España