El ciclo de vida de un sistema de información

Un sistema de información es un sistema, automatizado o manual, que engloba a personas, máquinas y/o métodos organizados para recopilar, procesar, transmitir datos que representan información. Un sistema de información engloba la infraestructura, la organización, el personal y todos los componentes necesarios para la recopilación, procesamiento, almacenamiento, transmisión, visualización, diseminación y organización de la información.

El ciclo de vida de un sistema de información

Las etapas del proceso de desarrollo de software

Cualquier sistema de información va pasando por una serie de fases a lo largo de su vida. Su ciclo de vida comprende una serie de etapas entre las que se encuentran las siguientes:

- Planificación
- Análisis
- Diseño
- Implementación
- Pruebas
- Instalación o despliegue
- Uso y mantenimiento

Estas etapas son un reflejo del proceso que se sigue a la hora de resolver cualquier tipo de problema. Ya en 1945, mucho antes de que existiese la Ingeniería del Software, el matemático George Polya describió este proceso en su libro *How to solve it* (el primero que describe la utilización de técnicas heurísticas en la resolución de problemas). Básicamente, resolver un problema requiere:

- Comprender el problema (análisis)
- Plantear una posible solución, considerando soluciones alternativas (diseño)
- Llevar a cabo la solución planteada (implementación)
- Comprobar que el resultado obtenido es correcto (pruebas)

Las etapas adicionales de planificación, instalación y mantenimiento que aparecen en el ciclo de vida de un sistema de información son necesarias en el mundo real porque el desarrollo de un sistema de información conlleva unos costes asociados (lo que se hace necesaria la planificación) y se supone que, una vez construido el sistema de información, éste debería poder utilizarse (si no, no tendría sentido haber invertido en su desarrollo).

Para cada una de las fases en que hemos descompuesto el ciclo de vida de un sistema de información se han propuesto multitud de prácticas útiles, entendiendo por prácticas aquellos

conceptos, principios, métodos y herramientas que facilitan la consecución de los objetivos de cada etapa.

En los párrafos siguientes se mencionan algunas de las actividades que han de realizarse en cada una de las fases del ciclo de vida de un sistema de información:

Planificación

Antes de que se le de oficialmente el pistoletazo de salida a un proyecto de desarrollo de un sistema de información, es necesario realizar una serie de tareas previas que influirán decisivamente en la finalización con éxito del proyecto. Estas tareas se conocen popularmente como el *fuzzy front-end* del proyecto al no estar sujetas a plazos. Las tareas iniciales que se realizarán esta fase inicial del proyecto incluyen actividades tales como la determinación del ámbito del proyecto, la realización de un estudio de viabilidad, el análisis de los riesgos asociados al proyecto, una estimación del coste del proyecto, su planificación temporal y la asignación de recursos a las distintas etapas del proyecto.

Delimitación del ámbito del proyecto

Resulta esencial determinar el ámbito del proyecto al comienzo del mismo. Han de establecerse de antemano qué cuestiones han de resolverse durante la realización del proyecto y cuáles se dejarán fuera. Tan importante es determinar los aspectos abarcados por el proyecto como fijar aquéllos aspectos que no se incluirán en el proyecto. Estos últimos han de indicarse explícitamente. Si es necesario, se puede especificar todo aquello que se posponga hasta una versión posterior del sistema. Si, en algún momento, fuese necesario incluir en el proyecto algún aspecto que no había sido considerado o que ya había sido descartado, es obligatorio reajustar la estimación del coste del proyecto y su planificación temporal.

Como resultado de la delimitación del ámbito del proyecto se obtiene un documento breve, de 1 ó 2 páginas, en el que se describe el problema que nuestro sistema de información pretende resolver. Este documento, denominado a veces *mission statement* o *project charter*, debe existir siempre en todo proyecto. En él se recogerá la descripción de más alto nivel de la funcionalidad que tendrá nuestro sistema de información, sus características principales y sus objetivos clave. Obviamente, este documento debe formar parte del contrato que se firme con el cliente en el arranque oficial del proyecto.

Además de ser breve, una buena descripción del proyecto debe superar con éxito la *prueba del ascensor*. Debe estar escrito en un lenguaje que cualquiera pueda entender, evitando un vocabulario excesivamente técnico. Además, debe recoger todo lo que le contaríamos a un conocido en unos segundos acerca del proyecto en el que estamos trabajando si nos lo cruzáramos por la calle o nos lo encontrásemos en un ascensor.

Estudio de viabilidad

Con recursos ilimitados (tiempo y dinero), casi cualquier proyecto se podría llevar a buen puerto. Por desgracia, en la vida real los recursos son más bien escasos, por lo que no todos los proyectos son viables. En un conocido informe de 1994 (el informe *Chaos* del *Standish Group*), se hizo un estudio para determinar el alcance de la conocida como "crisis crónica de la programación" y, en la medida de lo posible, identificar los principales factores que hacen fracasar proyectos de desarrollo de software y los ingredientes clave que pueden ayudar a reducir el índice de fracasos. De entre los proyectos analizados:

- Sólo uno de cada seis se completó a tiempo, de acuerdo con su presupuesto y con todas las características inicialmente especificadas.

- La mitad de los proyectos llegó a completarse eventualmente, costando más de lo previsto, tardando más tiempo del estimado inicialmente y con menos características de las especificadas al comienzo del proyecto.

- Por último, más de un 30% de los proyectos se canceló antes de completarse.

Dado que cinco de cada seis proyectos analizados no se ajustaron al plan previsto, no es de extrañar que resulte aconsejable realizar un estudio de viabilidad antes de comenzar el desarrollo de un sistema de información para determinar si el proyecto es económica, técnicas y legalmente viable. De hecho, lo primero que deberíamos hacer es plantearnos si la mejor opción es desarrollar un sistema informatizado o es preferible un sistema manual. Algo así debieron hacer los rusos cuando decidieron llevar lápices al espacio (según dicen, los americanos gastaron una fortuna hasta que inventaron un bolígrafo que funcionaba en ausencia de gravedad).

Antes de comenzar un proyecto, se debería evaluar la viabilidad económica, técnica y legal del mismo. Y no sólo eso, el resultado del estudio de viabilidad debería ajustarse a la realidad. A Jerry Weinberg, un conocido consultor, se le ocurrió preguntar a los asistentes a una conferencia suya, en el Congreso Internacional de Ingeniería del Software de 1987, cuántos de ellos habían participado en un estudio de viabilidad en el que se hubiese determinado que el proyecto no era técnicamente viable. De los mil quinientos asistentes, nadie levantó la mano.

Análisis de riesgos

Independientemente de la precisión con la que hayamos preparado nuestro proyecto, siempre se produce algún contratiempo que eche por tierra la mejor de las planificaciones. Es algo inevitable con lo que hemos de vivir y para lo cual disponemos de una herramienta extremadamente útil: la gestión de riesgos, que tradicionalmente se descompone en evaluación de riesgos y control de riesgos.

La evaluación de riesgos se utiliza para identificar "riesgos" que pueden afectar negativamente al plan de nuestro proyecto, estimar la probabilidad de que el riesgo se materialice y analizar su posible impacto en nuestro proyecto. ¿Qué sucedería si algún miembro clave del nuestro equipo abandona la empresa, se va de vacaciones, se pone enfermo o pide una baja por depresión causada por un entorno de trabajo hostil? ¿y si al final nos encontramos con algún problema de compatibilidad del sistema que hemos desarrollamos con la configuración de los equipos sobre los que ha de funcionar? ¿si, inadvertidamente, borramos o modificamos erróneamente algún que otro fichero clave? ¿si nuestro ordenador se avería?

Una vez analizados los riesgos potencialmente más peligrosos, podemos recurrir a distintas técnicas de control de riesgos. Por ejemplo, podemos elaborar planes de contingencia para los riesgos que sean más probables y de consecuencias más desastrosas para el proyecto. O tal vez seamos capaces de eliminar el riesgo de raíz (o mitigarlo) si buscamos alguna alternativa en la que el riesgo identificado no pueda presentarse (o se presente debilitado). Independientemente de la solución por la que optemos, el análisis de riesgos nos enseña que hemos de dejar un margen para imprevistos previsibles y añadir cierta holgura a la planificación de nuestro proyecto. Las hipótesis barajadas al analizar riesgos potenciales pueden convertirse en realidad y nunca está de más dejar algo de margen de maniobra.

Estimación

Sin duda, una de las tareas más peliagudas de cualquier proyecto de desarrollo de software es la estimación inicial del coste de algo que aún no conocemos. De hecho, la realización de malas estimaciones ha sido identificada como una de las dos causas más comunes del fracaso de un proyecto de desarrollo de software (Glass, 2003). La otra es la existencia de requerimientos inestables sujetos a continuos cambios.

Como dijo Böhr, la predicción es difícil, especialmente si se trata del futuro. Además, la estimación del coste asociado se suele realizar en el peor momento posible: al comienzo, cuando menos conocemos del proyecto y mayor es el margen del error de la estimación. Afortunadamente, existen algunas reglas heurísticas que nos pueden ayudar a estimar con una precisión razonable el coste y duración de un proyecto. El arte de una buena estimación está basado, fundamentalmente, en la experiencia del estimador. Haber participado en proyectos de similares características puede ser esencial para que seamos capaces de realizar una buena estimación. También podemos obtener resultados aceptables si tenemos en cuenta lo siguiente:

- Nunca se ha de realizar una estimación sobre la marcha, por mucho que nos presionen. Una respuesta apresurada sólo sirve para pillarnos los dedos y que después no podamos cumplir con las expectativas que nosotros mismos hemos creado. Una estimación siempre ha de ser meditada, tras un estudio pormenorizado de los distintos factores que pueden afectar a la realización de nuestro proyecto.

- La incertidumbre en la estimación es inevitable, pero en ocasiones puede reducirse. Cuantos más datos históricos recopilemos y más precisa sea la información de la que dispongamos acerca de nuestro proyecto, mejor será nuestra estimación.

- Hemos de descomponer nuestro proyecto en tareas de la granularidad adecuada. El error que se comete al estimar el conjunto de las actividades del proyecto es mayor que el que se comete cuando estimamos cada una de las actividades por separado. Los errores que cometamos en las distintas estimaciones tenderán a compensarse (la ley de los grandes números en acción).

- Es muy frecuente subestimar el esfuerzo necesario cuando descomponemos un problema complejo en multitud de tareas. Esto se debe a que, durante el transcurso del proyecto, también han de realizarse otras muchas tareas que probablemente hayamos olvidado incluir en nuestra estimación. Además, consideradas de forma independiente, las distintas tareas del proyecto resultan aparentemente más fáciles de realizar de lo que en realidad son.

- Resulta aconsejable utilizar varias técnicas de estimación y contrastar los resultados con ellas obtenidos. Por ejemplo, podemos realizar una estimación en función del coste un proyecto similar, utilizar algún modelo matemático de estimación (COCOMO o similar) y realizar una tercera estimación descomponiendo nuestro proyecto en tareas. Si los resultados obtenidos con las distintas técnicas de estimación son similares, probablemente nuestra estimación sea buena.

Planificación temporal y asignación de recursos

Una vez que hemos decidido seguir adelante con nuestro proyecto, hemos de planificar su temporización. Una planificación excesivamente detallada (con el proyecto descompuesto en tareas de un día, por ejemplo) puede resultar contraproducente. Cualquier error de planificación causado por algún imprevisto nos forzará a replanificar el resto del proyecto, retrasando aún más nuestro proyecto. Una planificación por semanas suele ser razonable para afrontar con comodidad las contingencias con las que nos vayamos encontrando sin tener que estar continuamente reajustando el plan del proyecto.

Pase lo que pase, la planificación del proyecto ha de reajustarse cada vez que cambien las circunstancias del mismo. Si no se ha podido terminar una tarea en el tiempo inicialmente establecido, no nos vale suponer alegremente que posteriormente se recuperará el tiempo perdido. Los proyectos se retrasan poco a poco. Debemos aprovechar las primeras señales de alarma y no esconderlas debajo de la alfombra fingiendo que todo marcha según lo previsto.

Tampoco vale decir que algo está casi terminado. O lo está o no lo está. Si no lo está, el plan ha de ajustarse a la situación real del proyecto. Pensar que algo está casi acabado sólo sirve

para darnos una falsa sensación de seguridad. Además, el 10% final de algo tiende a requerir el 90% del tiempo, así que no nos sirve de nada saber que hemos realizado 80% del esfuerzo si no podemos asegurar a qué parte corresponde el 20% que nos queda (¿a la que se termina rápidamente o a la que consume la mayor parte de nuestro tiempo?). Como dice un viejo adagio: "la primera mitad se lleva el 90% del tiempo; la segunda, el 90% restante". No use nunca en su planificación el hecho de que determinadas actividades del proyecto están casi terminadas.

La planificación es fundamental en la gestión de un proyecto de desarrollo de software. Procure siempre mantener su plan al día. Un plan que no se ajusta a la realidad no sirve de mucho. Cuando algún retraso indique que posiblemente le será imposible cumplir los plazos establecidos, hable con su cliente. A él le interesa saberlo y, aunque probablemente no se lo agradezca, a la larga resultará beneficioso y usted habrá cumplido con su obligación profesional.

Algunos errores comunes

Cuando las cosas no marchen del todo bien, hemos de evitar tomar algunas medidas que lo único que conseguirán es perjudicarnos. Aquí van algunas de las más comunes:

- Abreviar las etapas iniciales del proceso de desarrollo de software (planificación y análisis, generalmente) para pasar directamente a la "construcción" del sistema. Los errores cometidos en las fases iniciales de un proyecto son mucho más costosos de corregir a la larga, por lo que abreviar las etapas iniciales tiene graves consecuencias.

- No gestionar adecuadamente los cambios que inevitablemente ocurren durante el proyecto. Tan malo es permitir cualquier cambio de forma indiscriminada como ser excesivamente rígidos a la hora de no admitir cambios aunque éstos sean razonables.

- Reducir la interacción con el cliente, ya que aparentemente sólo se dedica a entorpecer nuestro trabajo con sus continuos cambios de opinión y sus expectativas poco realistas. Craso error. Al fin y al cabo, el cliente es la persona cuyas necesidades hemos de descubrir y satisfacer.

- Olvidar que añadir personal a un proyecto retrasado, por lo general, sólo lo retrasa más (la ley de Brooks). La curva de aprendizaje que se necesita para comenzar a ser productivo ha de tenerse siempre en cuenta. Además, ¿quién le resuelve las dudas al recién incorporado? El tiempo que empleen los demás miembros del equipo será tiempo que no podrán utilizar en otras cosas.

- Someter a los miembros de nuestro equipo a continuas interrupciones durante su jornada de trabajo (llamadas telefónicas, reuniones, consultas...). Las calidad del trabajo intelectual depende de la capacidad del trabajador de mantener su "estado

Algunos errores comunes

de flujo" (un estado relajado de inmersión total en un problema que facilita su comprensión y la generación de soluciones). Se tarda unos 15 minutos en conseguir este estado, por lo que una simple interrupción cada 10 minutos afecta drásticamente al rendimiento del trabajador.

- Hacer trabajar horas extra a los miembros del equipo de desarrollo sólo sirve para disminuir su productividad (trabajo realizado por unidad de tiempo). Tras un larga jornada de trabajo, la mente pierde su frescura y comete errores que luego tendrán que corregirse. ¿Adivina cómo? Con más horas extra.

- No informar de pequeños retrasos pensando que más tarde se recuperará el tiempo perdido. La planificación temporal del proyecto debe ir ajustándose conforme vamos aprendiendo más cosas acerca del problema al que nos enfrentamos. En el peor de los casos, se deben negociar algunos recortes con el cliente si éste desea mantener los plazos estipulados al comienzo del proyecto.

- Confiar excesivamente en la mejora de rendimiento que se producirá gracias al uso de una nueva herramienta, tecnología o metodología (error conocido como el "síndrome de la bala de plata" en honor a un artículo muy recomendable escrito por Fred Brooks: *No silver bullets - Essence and accidents of Software Engineering*, *Computer*, 1987).

Observe el tipo de errores comunes que aparece en la lista anterior. En general, no prestar la debida atención a las necesidades de los integrantes del equipo de desarrollo o a las del cliente garantiza que un proyecto no terminará con éxito. El factor humano es más importante que el técnico.

Análisis

Lo primero que debemos hacer para construir un sistema de información es averiguar qué es exactamente lo que tiene que hacer el sistema. La etapa de análisis en el ciclo de vida del software corresponde al proceso mediante el cual se intenta descubrir qué es lo que realmente se necesita y se llega a una comprensión adecuada de los requerimientos del sistema (las características que el sistema debe poseer).

¿Por qué resulta esencial la etapa de análisis? Simplemente, porque si no sabemos con precisión qué es lo que se necesita, ningún proceso de desarrollo nos permitirá obtenerlo. El problema es que, de primeras, puede que ni nuestro cliente sepa de primeras qué es exactamente lo que necesita. Por tanto, deberemos ayudarle a averiguarlo con ayuda de distintas técnicas (algunas de las cuales aprenderemos a utilizar más adelante).

¿Por qué es tan importante averiguar exactamente cuáles son los requerimientos del sistema si el software es fácilmente maleable (aparentemente)? Porque el coste de construir correctamente un sistema de información a la primera es mucho menor que el coste de construir un sistema que habrá que modificar más adelante. Cuanto antes se detecte un error, mejor. Distintos estudios han demostrado que eliminar un error en las fases iniciales de un proyecto (en la etapa de análisis) resulta de 10 a 100 veces más económico que subsanarlo al final del proyecto. Conforme avanza el proyecto, el software se va describiendo con un mayor nivel de detalle, se concreta cada vez más y se convierte en algo cada vez más rígido.

¿Es posible determinar de antemano todos los requerimientos de un sistema de información? Desgraciadamente, no. De hecho, una de las dos causas más comunes de fracaso en proyectos de desarrollo de software es la inestabilidad de los requerimientos del sistema (la otra es una mala estimación del esfuerzo requerido por el proyecto). En el caso de una mala estimación, el problema se puede solucionar estableciendo objetivos más realistas. Sin embargo, en las etapas iniciales de un proyecto, no disponemos de la información necesaria para determinar exactamente el problema que pretendemos resolver. Por mucho tiempo que le dediquemos al análisis del problema (un fenómeno conocido como la parálisis del análisis).

La inestabilidad de los requerimientos de un sistema es inevitable. Se estima que un 25% de los requerimientos iniciales de un sistema cambian antes de que el sistema comience a utilizarse. Muchas prácticas resultan efectivas para gestionar adecuadamente los requerimientos de un sistema y, en cierto modo, controlar su evolución. Un buen analista debería tener una formación adecuada en:

- Técnicas de elicitación de requerimientos.
- Herramientas de modelado de sistemas.
- Metodologías de análisis de requerimientos.

Técnicas de elicitación de requerimientos

En la fase de análisis, los errores más difíciles de corregir son los causados por "requerimientos ausentes", generalmente en la forma de suposiciones que se dan por sabidas pero nunca se llegan a plasmar explícitamente. Por este motivo, elicitar los requerimientos de un sistema de información (esto es, obtener de algún modo cuáles son realmente esos requerimientos) resulta una actividad esencial en cualquier proceso de desarrollo de software.

La elicitación de requerimientos requiere previamente la identificación de las personas afectadas por el proyecto, sus *stakeholders* (literalmente, los que apuestan algo), lo que incluye desde el cliente que paga el proyecto hasta los usuarios finales de la aplicación, sin olvidarse de terceras personas y organizaciones relacionadas indirectamente con el sistema que se va a desarrollar (por ejemplo, empresas competidoras y organismos reguladores).

En la elicitación de requerimientos se recurre a distintas técnicas que favorezcan la comunicación entre el analista y el resto de personas involucradas, como puede ser la realización de entrevistas (en las que importa no sólo lo que se pregunta, sino cómo se pregunta), el diseño de cuestionarios (cuando no tenemos tiempo ni recursos para entrevistar personalmente a todo el mundo) o el desarrollo de prototipos (para recoger información que, de otra forma, no obtendríamos hasta las etapas finales del proyecto, cuando cualquier rectificación saldría mucho más cara). También se puede observar el funcionamiento normal del entorno en el que se instalará nuestro sistema o, incluso, participar activamente en él (por ejemplo, desempeñando temporalmente el trabajo de los usuarios de nuestro sistema). Por último, también podemos investigar por nuestra cuenta consultando documentos relacionados con el tema de nuestro proyecto o estudiando productos similares que ya existan en el mercado.

Herramientas de modelado de sistemas

Un modelo, básicamente, no es más que una simplificación de la realidad. El uso de modelos en la construcción de sistemas de información resulta esencial por los siguientes motivos:

- Los modelos ayudan a comunicar la estructura de un sistema complejo (y, por tanto, a comunicarnos con las demás personas involucradas en un proyecto).
- Los modelos sirven para especificar el comportamiento deseado del sistema (como guía para las etapas posteriores del proyecto).
- Los modelos nos ayudan a comprender mejor lo que estamos diseñando (por ejemplo, para detectar inconsistencias y corregirlas).
- Los modelos nos permiten descubrir oportunidades de simplificación (ahorrarnos trabajo en el proyecto actual) y de reutilización (ahorrarnos trabajo en futuros proyectos).

En resumidas cuentas, los modelos, entre otras cosas, facilitan el análisis de los requerimientos del sistema, así como su posterior diseño e implementación. Un modelo, en definitiva, proporciona "los planos" de un sistema. El modelo ha de capturar "lo esencial" desde determinado punto de vista. En función de para qué queramos el modelo, lo haremos más o menos detallado, siempre de acuerdo a su relevancia y utilidad.

Un sistema de información es un sistema complejo, por lo que a (casi) nadie se le ocurriría intentar describirlo utilizando un único modelo. De hecho, todo sistema puede describirse desde distintos puntos de vista y nosotros utilizaremos distintos tipos de modelos dependiendo del aspecto del sistema en que deseemos centrar nuestra atención:

- Existen **modelos estructurales** que nos ayudan a la hora de organizar un sistema complejo. Por ejemplo, un diagrama entidad/relación nos indica cómo se

estructuran los datos de un sistema de información, mientras que un diagrama de flujo de datos nos da información acerca de cómo se descompone un sistema en subsistemas y del flujo de datos que existe entre los distintos subsistemas.

- También existen **modelos de comportamiento** que nos permiten analizar y modelar la dinámica de un sistema. Por ejemplo, un diagrama de estados representa los distintos estados en que puede encontrarse un sistema y cómo se puede pasar de un estado a otro, mientras que la descripción de un caso de uso nos ayuda a comprender la secuencia de pasos involucrada en la consecución de un objetivo concreto por parte de un usuario del sistema.

Más adelante aprenderemos las notaciones asociadas a distintos tipos de modelos cuya utilización resulta recomendable en el diseño de un sistema de información. Aprenderemos su significado, veremos su utilidad y ofreceremos algunos consejos heurísticos acerca de su correcto uso.

Metodologías de análisis de requerimientos

Las técnicas de elicitación de requerimientos y las herramientas de modelado de sistemas de las que hemos hablado en los párrafos anteriores deben utilizarse acompañadas de una metodología adecuada. En este contexto, una metodología no es más que un conjunto de convenciones que han resultado útiles en la práctica y cuyo uso combinado se recomienda.

Las metodologías de análisis particulares, de las que hay muchas, usualmente están ligadas, o bien al uso de determinadas herramientas (por lo que el vendedor de la herramienta se convierte, muchas veces, en el único promotor de la metodología), o bien a empresas de consultoría concretas (que ofrecen cursos de aprendizaje de la metodología que proponen).

En general, no obstante, la elección adecuada de las técnicas utilizadas dependerá de la situación concreta en la que se encuentre nuestro proyecto. Por este motivo, lo más adecuado es aprender cuantas más técnicas mejor y averiguar en qué situaciones resulta más efectiva cada una de ellas.

Diseño

Mientras que los modelos utilizados en la etapa de análisis representan los requisitos del usuario desde distintos puntos de vista (el qué), los modelos que se utilizan en la fase de diseño representan las características del sistema que nos permitirán implementarlo de forma efectiva (el cómo).

Un software bien diseñado debe exhibir determinadas características. Su diseño debería ser modular en vez de monolítico. Sus módulos deberían ser cohesivos (encargarse de una tarea

concreta y sólo de una) y estar débilmente acoplados entre sí (para facilitar el mantenimiento del sistema). Cada módulo debería ofrecer a los demás unos interfaces bien definidos (al estilo del diseño por contrato propuesto por Bertrand Meyer) y ocultar sus detalles de implementación (siguiendo el principio de ocultación de información de Parnas). Por último, debe ser posible relacionar las decisiones de diseño tomadas con los requerimientos del sistema que las ocasionaron (algo que se suele denominar "trazabilidad de los requerimientos").

En la fase de diseño se han de estudiar posibles alternativas de implementación para el sistema de información que hemos de construir y se ha de decidir la estructura general que tendrá el sistema (su **diseño arquitectónico**). El diseño de un sistema es complejo y el proceso de diseño ha de realizarse de forma iterativa. La solución inicial que propongamos probablemente no resulte la más adecuada para nuestro sistema de información, por lo que deberemos refinarla. Afortunadamente, tampoco es necesario que empecemos desde cero. Existen auténticos catálogos de patrones de diseño que nos pueden servir para aprender de los errores que otros han cometido sin que nosotros tengamos que repetirlos.

Igual que en la etapa de análisis creábamos distintos modelos en función del aspecto del sistema en que centrábamos nuestra atención, el diseño de un sistema de información también presenta distintas **facetas**:

- Por un lado, es necesario abordar el **diseño de la base de datos**, un tema que trataremos detalladamente más adelante.

- Por otro lado, también hay que diseñar las **aplicaciones** que permitirán al usuario utilizar el sistema de información. Tendremos que diseñar la interfaz de usuario del sistema y los distintos componentes en que se descomponen las aplicaciones. De esto último hablaremos en las dos secciones siguientes.

Arquitecturas multicapa

La división de un sistema en distintas capas o niveles de abstracción es una de las técnicas más comunes empleadas para construir sistemas complejos. Esta división se puede apreciar en el hardware, donde el diseño de un sistema en un lenguaje de alto nivel como VHDL o Verilog se traduce en un diseño a nivel de registros lógicos (RTL); éste se implementa mediante puertas lógicas, a partir de las cuales se obtiene un diseño a nivel de transistores; los transistores, finalmente, se crean en un circuito integrado con una serie de máscaras. Los protocolos de red también se diseñan utilizando distintas capas: la capa de aplicación (HTTP) utiliza los servicios de la capa de transporte (TCP), la cual se implementa sobre la capa de red (IP) y así sucesivamente hasta llegar a la transmisión física de los datos a través de algún medio de transmisión.

En realidad, el uso de capas es una forma más de la técnica de resolución de problemas conocida con el nombre de "divide y vencerás", que se basa en descomponer un problema

complejo en una serie de problemas más sencillos de forma que se pueda obtener la solución al problema complejo a partir de las soluciones a los problemas más sencillos. Al dividir un sistema en capas, cada capa puede tratarse de forma independiente (sin tener que conocer los detalles de las demás).

Desde el punto de vista de la Ingeniería del Software, la división de un sistema en capas facilita el diseño modular (cada capa encapsula un aspecto concreto del sistema) y permite la construcción de sistemas débilmente acoplados (si minimizamos las dependencias entre capas, resultará más fácil sustituir la implementación de una capa sin afectar al resto del sistema). Además, el uso de capas también fomenta la reutilización (p.ej. TCP/IP se utiliza en una amplia variedad de aplicaciones, desde HTTP y FTP hasta telnet y SSH).

Como es lógico, la parte más difícil en la construcción de un sistema multicapa es decidir cuántas capas utilizar y qué responsabilidades asignarle a cada capa.

En las **arquiecturas cliente/servidor** se suelen utilizar dos capas. En el caso de las aplicaciones informáticas de gestión, esto se suele traducir en un servidor de bases de datos en el que se almacenan los datos y una aplicación cliente que contiene la interfaz de usuario y la lógica de la aplicación.

El problema con esta descomposición es que la lógica de la aplicación suele acabar mezclada con los detalles de la interfaz de usuario, dificultando las tareas de mantenimiento a que todo software se ve sometido y destruyendo casi por completo la portabilidad del sistema, que queda ligado de por vida a la plataforma para la que se diseñó su interfaz en un primer momento.

Mantener la misma arquitectura y pasar la lógica de la aplicación al servidor tampoco resulta una solución demasiado acertada. Se puede implementar la lógica de la aplicación utilizando procedimientos almacenados, pero éstos suelen tener que implementarse en lenguajes estructurados no demasiado versátiles. Además, suelen ser lenguajes específicos para cada tipo de base de datos, por lo que la portabilidad del sistema se ve gravemente afectada.

La solución, por tanto, pasa por crear nueva capa en la que se separe la lógica de la aplicación de la interfaz de usuario y del mecanismo utilizado para el almacenamiento de datos. El sistema resultante tiene tres capas:

- La capa de **presentación**, encargada de interactuar con el usuario de la aplicación mediante una interfaz de usuario (ya sea una interfaz web, una interfaz Windows o una interfaz en línea de comandos, aunque esto último suele ser menos habitual en la actualidad).

- La **lógica de la aplicación** [a la que se suele hacer referencia como *business logic* o *domain logic*], usualmente implementada utilizando un modelo orientado a objetos del dominio de la aplicación, es la responsable de realizar las tareas para las cuales se diseña el sistema.

- La capa de **acceso a los datos**, encargada de gestionar el almacenamiento de los datos, generalmente en un sistema gestor de bases de datos relacionales, y de la comunicación del sistema con cualquier otro sistema que realice tareas auxiliares (p.ej. middleware).

Cuando el usuario del sistema no es un usuario humano, se hace evidente la similitud entre las capas de presentación y de acceso a los datos. Teniendo esto en cuenta, el sistema puede verse como un núcleo (lógica de la aplicación) en torno al cual se crean una serie de interfaces con entidades externas. Esta vista simétrica del sistema es la base de la *arquitectura hexagonal* de Alistair Cockburn.

No obstante, aunque sólo fuese por las peculiaridades del diseño de interfaces de usuario, resulta útil mantener la vista asimétrica del software como un sistema formado por tres capas. Por ejemplo, la interfaz de usuario debe permitir que el usuario se pueda equivocar (y rectificar de la manera menos traumática posible) y estar especialmente diseñada para agilizar su trabajo (y nunca entorpecerlo). Además, suele ser recomendable diferenciar lo que se suministra (presentación) de lo que se consume (acceso a los servicios suministrados por otros sistemas).

A pesar del atractivo de esta arquitectura con tres capas (basta con pensar lo que facilitaría la conversión de aplicaciones Windows en aplicaciones web), esta arquitectura no se ha impuesto del todo porque las herramientas de desarrollo suelen estar diseñadas para construir aplicaciones cliente/servidor ligadas a algún productor de software. De hecho, puede resultar difícil (e incluso imposible) descomponer un sistema en tres capas con determinadas herramientas de desarrollo.

Notas acerca del diseño de las aplicaciones

Si suponemos que hemos sido capaces de separar el núcleo de la aplicación de sus distintos interfaces, aún nos queda por decidir cómo vamos a organizar la implementación de la lógica asociada a la aplicación. Como en cualquier otra tarea de diseño, tenemos que llegar a un compromiso adecuado entre distintos intereses. Por un lado, nos gustaría que el diseño resultante fuese lo más sencillo posible. Por otro lado, sería deseable que nuestro diseño estuviese bien preparado para soportar las modificaciones que hayan de realizarse en el futuro.

Por lo general, el diseño de la lógica una aplicación se suele ajustar a uno de los tres siguientes patrones de diseño:

- **Rutinas**: La forma más simple de implementar cualquier sistema se basa en implementar procedimientos y funciones que acepten y validen las entradas recibidas de la capa de presentación, realicen los cálculos necesarios, utilicen los servicios de aquellos sistemas que hagan falta para completar la operación,

almacenen los datos en las bases de datos y envíen una respuesta adecuada al usuario. Básicamente, cada acción que el usuario pueda realizar se traducirá en un procedimiento que realizará todo lo que sea necesario al más puro estilo del diseño estructurado tradicional. Aunque este modelo sea simple y pueda resultar adecuado a pequeña escala, la evolución de las aplicaciones diseñadas de esta forma suele acabar siendo una pesadilla para las personas encargadas de su mantenimiento.

- **Módulos de datos**: Ante la situación descrita en el párrafo anterior, lo usual es dividir el sistema utilizando los distintos conjuntos de datos con los que trabaja la aplicación para crear módulos más o menos independientes. De esta forma, se facilita la eliminación de lógica duplicada. De hecho, muchos de los entornos de desarrollo visual de aplicaciones permiten definir módulos de datos que encapsulen los conjuntos de datos con los que se trabaja y la lógica asociada a ellos. Las herramientas de Borland, Delphi y C++Builder, son un claro ejemplo, igual que el Developer de Oracle. Microsoft, en su arquitectura DNA [Distributed interNet Application], fomenta este estilo al emplear conjuntos de datos (resultado de ejecutar consultas SQL) sobre los cuales operan directamente las distintas capas de una aplicación multicapa. En el caso de la plataforma .NET, la clase `DataSet` proporciona la base sobre la que se montaría todo el diseño de una aplicación (algo que obviamente facilita el Visual Studio .NET).

- **Modelo del dominio**: Una tercera opción, la ideal para cualquier purista de la orientación a objetos, es crear un modelo orientado a objetos del dominio de la aplicación. En vez de que una rutina se encargue de todo lo que haya que hacer para completar una acción, cada objeto es responsable de realizar las tareas que le atañen directamente.

En la práctica, no todo es blanco o negro. Aunque empleemos un modelo orientado a objetos del dominio de la aplicación, es habitual crear una capa intermedia entre la capa de presentación y la lógica de la aplicación a la que se suele denominar **capa de servicio**. La interfaz de la capa de servicio incluirá métodos asociados a las distintas acciones que pueda realizar el usuario, si bien, en vez de incluir en ella la lógica de la aplicación, la capa de servicio delega inmediatamente en los objetos responsables de cada tarea. En cierto modo, la capa de servicio se encarga de la lógica específica de la aplicación, dejando para el modelo del dominio la lógica del dominio (común a cualquier aplicación que se construya sobre el mismo dominio de aplicación).

Cualquier aplicación sufre modificaciones de mayor o menor importancia a lo largo de su vida útil. Dichas modificaciones alteran el diseño inicial de la aplicación y tienden a aumentar su entropía. Si utilizamos rutinas para implementar la lógica de nuestra aplicación en el sentido tradicional, las modificaciones pueden suponer tener que revisar el código completo de la aplicación para descubrir lo que hay que actualizar. Es como si tuviésemos una habitación completamente desordenada en la que hay que encontrar algo en particular. En el caso de los módulos de datos, el impacto de las modificaciones suele ser más fácil de

determinar pero, aún así, podemos encontrarnos con sorpresas desagradables si todos los módulos de nuestra aplicación trabajan sobre un conjunto de datos cuya estructura debemos alterar ligeramente. Por último, si diseñamos un buen modelo orientado a objetos, la encapsulación proporcionada por los objetos de nuestra aplicación permitirá que el impacto de las modificaciones sea de carácter local en la mayoría de las ocasiones. En cierto modo, estamos limitando el aumento de la entropía al interior de los cajones (algo que la mayoría de nosotros tolera sin demasiados problemas).

Sobre el acceso a los datos

Existen distintas formas en las que una aplicación implementada utilizando un lenguaje de programación de propósito general puede acceder a los datos, almacenados por lo general en una base de datos relacional:

- La primera opción que se nos puede ocurrir cuando diseñamos un sistema orientado a objetos es utilizar un **registros activos**, objetos que encapsulan directamente las estructuras de datos externas (p.ej. las tuplas de las tablas de la base de datos) e incorporan la lógica del dominio que les corresponda, aparte de las operaciones necesarias para obtener y guardar objetos en la base de datos.

- Algo más adecuado puede resultar el empleo de **gateways**, clases auxiliares que se corresponden con las tablas de la base de datos e implementan las operaciones necesarias para manipular la base de datos [CRUD: Create, Retrieve, Update & Delete]. Estas clases auxiliares nos permiten no mezclar la lógica de la aplicación con el acceso a los datos externos, tal como sucede si utilizamos *registros activos*.

- La tercera opción (y siempre hay una tercera opción) es la más compleja pero la más flexible: **O/R Mapping**. Se basa en establecer una correspondencia entre el modelo orientado a objetos del dominio y la representación de los distintos objetos en una base de datos relacional. En las dos alternativas anteriores, los objetos de la aplicación han de ser conscientes de cómo se representan en la base de datos. En el caso del *O/R Mapping*, los objetos pueden ignorar la estructura de la base de datos y cómo se realiza la comunicación con la base de datos. La inversión de control característica de esta opción independiza el modelo orientado a objetos del dominio de la capa de acceso a los datos: se puede cambiar la base de datos sin tener que tocar el modelo orientado a objetos del dominio y viceversa. De esta forma, se facilita el desarrollo, la depuración y la evolución de las aplicaciones.

Cuando para implementar la aplicación se utiliza una herramienta de programación visual (tipo Visual Studio .NET, C++Builder, Delphi o Developer), generalmente no podemos elegir la forma en que nuestra aplicación accederá a los datos. Pese a ello, siempre existe cierto margen de maniobra que podemos aprovechar para diseñar nuestro sistema lo mejor posible.

Aviso

Las etapas posteriores del ciclo de vida de un sistema de información (implementación, pruebas, despliegue, uso y mantenimiento) se describen a continuación con menor grado de detalle que las anteriores (planificación, análisis y diseño) porque su influencia es significativamente menor sobre el proceso de diseño de bases de datos.

Implementación

Una vez que sabemos qué funciones debe desempeñar nuestro sistema de información (análisis) y hemos decidido cómo vamos a organizar sus distintos componentes (diseño), es el momento de pasar a la etapa de implementación, pero nunca antes. Antes de escribir una sola línea de código (o de crear una tabla en nuestra base de datos) es fundamental haber comprendido bien el problema que se pretende resolver y haber aplicado principios básicos de diseño que nos permitan construir un sistema de información de calidad.

Para la fase de implementación hemos de seleccionar las herramientas adecuadas, un entorno de desarrollo que facilite nuestro trabajo y un lenguaje de programación apropiado para el tipo de sistema que vayamos a construir. La elección de estas herramientas dependerá en gran parte de las decisiones de diseño que hayamos tomado hasta el momento y del entorno en el que nuestro sistema deberá funcionar.

A la hora de programar, deberemos procurar que nuestro código no resulte indescifrable. Para que nuestro código sea legible, hemos de evitar estructuras de control no estructuradas, elegir cuidadosamente los identificadores de nuestras variables, seleccionar algoritmos y estructuras de datos adecuadas para nuestro problema, mantener la lógica de nuestra aplicación lo más sencilla posible, comentar adecuadamente el texto de nuestros programas y, por último, facilitar la interpretación visual de nuestro código mediante el uso de sangrías y líneas en blanco que separen distintos bloques de código.

Además de las tareas de programación asociadas a los distintos componentes de nuestro sistema, en la fase de implementación también hemos de encargarnos de la adquisición de todos los recursos necesarios para que el sistema funcione (por ejemplo, las licencias de uso del sistema gestor de bases de datos que vayamos a utilizar). Usualmente, también desarrollaremos algunos casos de prueba que nos permitan ir comprobando el funcionamiento de nuestro sistema conforme vamos construyéndolo.

Control de versiones

En todo proyecto de desarrollo de software resulta fundamental una adecuada gestión de la configuración del software (SCM, *Software Configuration Management*), más conocida vulgarmente por uno de sus aspectos, el control de versiones. De hecho, es una actividad clave en el nivel 2 del CMM (*Capability Maturity Model*), un modelo de madurez del proceso de desarrollo del software en el cual el nivel 1 representa la anarquía.

Independientemente de su importancia en el control del proceso de desarrollo de software (algo innegable), su valor es incalculable para evitar pérdidas irreparables (siempre y cuando se hagan copias de seguridad de la base de datos de nuestra herramienta de control de versiones) y también para volver cómodamente a una versión anterior de nuestro código si nuestras últimas modificaciones del código no resultaron del todo acertadas.

Pruebas

Errar es humano y la etapa de pruebas tiene como objetivo detectar los errores que se hayan podido cometer en las etapas anteriores del proyecto (y, eventualmente, corregirlos). Lo suyo, además, es hacerlo antes de que el usuario final del sistema los tenga que sufrir. De hecho, una prueba es un éxito cuando se detecta un error (y no al revés, como nos gustaría pensar).

La búsqueda de errores que se realiza en la etapa de pruebas puede adaptar distintas formas, en función del contexto y de la fase del proyecto en la que nos encontremos:

- Las **pruebas de unidad** sirven para comprobar el correcto funcionamiento de un componente concreto de nuestro sistema. Es este tipo de pruebas, el "probador" debe buscar situaciones límite que expongan las limitaciones de la implementación del componente, ya sea tratando éste como una caja negra ("pruebas de caja negra") o fijándonos en su estructura interna ("pruebas de caja blanca"). Resulta recomendable que, conforme vamos añadiéndole nueva funcionalidad a nuestras aplicaciones, vayamos creando nuevos tests con los medir nuestro progreso y también repitamos los antiguos para comprobar que lo que antes funcionaba sigue funcionando (*test de regresión*).

- Las **pruebas de integración** son las que se realizan cuando vamos juntando los componentes que conforman nuestro sistema y sirven para detectar errores en sus interfaces. En algunas empresas, como Microsoft, se hace una compilación diaria utilizando los componentes del sistema tal como estén en ese momento (*daily build*) y se somete al sistema a una serie de pruebas básicas (la prueba de humo, *smoke test*) que garanticen que el proyecto podrá seguir avanzando al día

siguiente. El causante de que la compilación diaria falle suele tener que quedarse a hacer horas extra para que sus compañeros puedan seguir trabajando al día siguiente...

- Una vez "finalizado" el sistema, se realizan **pruebas alfa** en el seno de la organización encargada del desarrollo del sistema. Estas pruebas, realizadas desde el punto de vista de un usuario final, pueden ayudar a pulir aspectos de la interfaz de usuario del sistema

- Cuando el sistema no es un producto a medida, sino que se venderá como un producto en el mercado, también se suelen realizar **pruebas beta**. Estas pruebas las hacen usuarios finales del sistema ajenos al equipo de desarrollo y pueden resultar vitales para que un producto tenga éxito en el mercado.

- En sistemas a medida, se suele realizar un **test de aceptación** que, si se supera con éxito, marcará oficialmente el final del proceso de desarrollo y el comienzo de la etapa de mantenimiento.

- Por último, a lo largo de todo el ciclo de vida del software, se suelen hacer **revisiones** de todos los productos generados a lo largo del proyecto, desde el documento de especificación de requerimientos hasta el código de los distintos módulos de una aplicación. Estas revisiones, de carácter más o menos formal, ayuden a verificar la corrección del producto revisado y también a validarlo (comprobar que se ajusta a los requerimientos reales del sistema).

Aunque es imposible garantizar la ausencia de errores en el software, una adecuada combinación de distintas técnicas de prueba puede ayudar más del 90% de los errores que se encontrarán a lo largo de toda la vida del sistema. Aunque podamos ser reacios a admitirlo, lo normal es que el 40% de nuestro tiempo lo "perdamos" eliminando errores, mientras que sólo empleamos un 20% en la etapa de análisis, otro 20% en el diseño y el 20% restante en la implementación del sistema (Robert Glass, *Building quality software*, 1992).

Al realizar cualquiera de los tipos de prueba descritos, es importante recordar que el desarrollo de software es una actividad que se realiza en equipo, por lo que pueden surgir roces personales y disputas políticas entre los miembros del equipo. Las pruebas resultan particularmente delicadas en este sentido, ya que su objetivo es, al fin y al cabo, encontrar defectos.

Instalación / Despliegue

Una vez concluidas las etapas de desarrollo de un sistema de información (análisis, diseño, implementación y pruebas), llega el instante de que poner el sistema en funcionamiento, su instalación o despliegue.

De cara a su instalación, hemos de planificar el entorno en el que el sistema debe funcionar, tanto hardware como software: equipos necesarios y su configuración física, redes de interconexión entre los equipos y de acceso a sistemas externos, sistemas operativos (actualizados para evitar problemas de seguridad), bibliotecas y componentes suministrados por terceras partes, etcétera.

Para asegurar el correcto funcionamiento del sistema, resulta esencial que tengamos en cuenta las dependencias que pueden existir entre los distintos componentes del sistema y sus versiones. Una aplicación puede que sólo funcione con una versión concreta de una biblioteca auxiliar. Un disco duro puede que sólo rinda al nivel deseado si instalamos un controlador concreto. Componentes que por separado funcionarían correctamente, combinados causan problemas, por lo que deberemos utilizar sólo combinaciones conocidas que no presenten problemas de compatibilidad.

Si nuestro sistema reemplaza a un sistema anterior o se despliega paulatinamente en distintas fases, también hemos de planificar cuidadosamente la transición del sistema antiguo al nuevo de forma que sus usuarios no sufran una disrupción en el funcionamiento del sistema. En ocasiones, el sistema se instala físicamente en un entorno duplicado y la transición se hace de forma instantánea una vez que la nueva configuración funciona correctamente. Cuando el presupuesto no da para tanto, tal vez haya que buscar un momento de baja utilización del sistema para realizar la actualización (por la noches o en fin de semana, por ejemplo).

Uso y mantenimiento

La etapa de mantenimiento consume típicamente del 40 al 80 por ciento de los recursos de una empresa de desarrollo de software. De hecho, con un 60% de media, es probablemente la etapa más importante del ciclo de vida del software. Dada la naturaleza del software, que ni se rompe ni se desgasta con el uso, su mantenimiento incluye tres facetas diferentes:

- Eliminar los defectos que se detecten durante su vida útil (**mantenimiento correctivo**), lo primero que a uno se le viene a la cabeza cuando piensa en el mantenimiento de cualquier cosa.

- Adaptarlo a nuevas necesidades (**mantenimiento adaptativo**), cuando el sistema ha de funcionar sobre una nueva versión del sistema operativo o en un entorno hardware diferente, por ejemplo.

- Añadirle nueva funcionalidad (**mantenimiento perfectivo**), cuando se proponen características deseables que supondrían una mejora del sistema ya existente.

De las distintas facetas del mantenimiento, la eliminación de defectos sólo supone el 17% del coste de mantenimiento de un sistema, mientras que el diseño e implementación de mejoras es responsable del 60% del coste de mantenimiento. Es decir, más de un tercio del coste total del

software se emplea en añadirle características a software ya existente (el 60% del 60%). La corrección de errores supone, en contraste, "sólo" en torno al 10% del coste total del software. Aún menos cuanto mejores sean las técnicas usadas en su desarrollo.

Se ha observado que, cuanto mejor sea el software, más tendremos que invertir en su mantenimiento, aun cuando se emplee menos esfuerzo en corregir defectos. Este hecho, que puede parecer paradójico, se debe, simplemente, a que nuestro sistema se usará más (a veces, de formas que no habíamos previsto). Por tanto, nos llegarán más propuestas de modificación y mejora que si el sistema hubiese quedado aparcado, cogiendo polvo, en algún rincón.

Si examinamos las tareas que se llevan a cabo durante la etapa de mantenimiento, nos encontramos que en el mantenimiento se repiten todas las etapas que ya hemos visto del ciclo de vida de un sistema de información. Al tratar principalmente de cómo añadirle nueva funcionalidad a un sistema ya existente, el mantenimiento repite "en miniatura" el ciclo de vida completo de un sistema de información. Es más, a las tareas normales de desarrollo hemos de añadirle una nueva, comprender el sistema que ya existe, por lo que se podría decir que el mantenimiento de un sistema es más difícil que su desarrollo (Glass, 2003).

Modelos de ciclo de vida

Todas las actividades descritas en las distintas secciones del apartado anterior están presentes en cualquier proyecto de desarrollo de software (además de otras muchas relativas a la gestión de un proyecto o a su control de calidad). Sin embargo, las tareas concretas que se realicen (y su grado de rigor) dependerán de la naturaleza del proyecto al que nos enfrentemos y de las características de nuestro entorno de trabajo.

El director de un proyecto, contando con el asesoramiento de los demás miembros del equipo, debe elegir los métodos y herramientas más adecuados en cada momento para satisfacer las necesidades específicas del proyecto, además de establecer las medidas oportunas que permitan controlar la evolución del proyecto. Las decisiones tomadas en este sentido han de tener como objetivo satisfacer los tiempos de entrega pactados con el cliente sin comprometer la calidad del producto final.

Existen distintas formas de organizar el orden concreto en el que se acometerán las distintas etapas del ciclo de vida de un sistema de información. En los siguientes párrafos se describen algunas de las alternativas que deberían tenerse en cuenta:

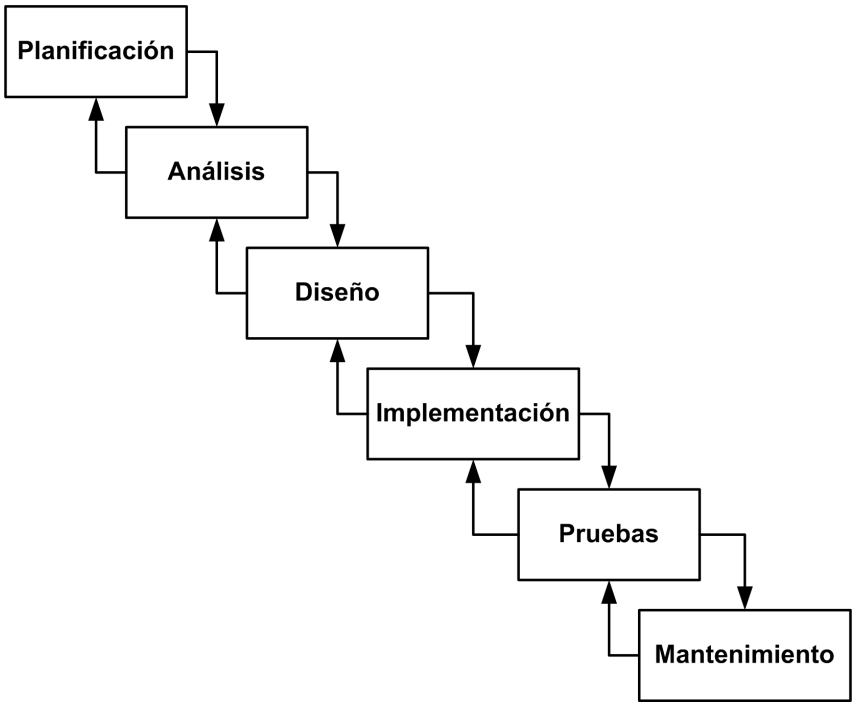
Ciclo de vida clásico

El modelo de ciclo de vida clásico, también denominado "modelo en cascada", se basa en intentar hacer las cosas bien desde el principio, de una vez y para siempre. Se pasa, en orden, de una etapa a la siguiente sólo tras finalizar con éxito las tareas de verificación y validación propias de la etapa. Si resulta necesario, únicamente se da marcha atrás hasta la fase inmediatamente anterior.

Este modelo tradicional de ciclo de vida exige una aproximación secuencial al proceso de desarrollo del software. Por desgracia, esta aproximación presenta una serie de graves inconvenientes, entre los que cabe destacar:

- Los proyectos reales raramente siguen el flujo secuencial de actividades que propone este modelo.
- Normalmente, es difícil para el cliente establecer explícitamente todos los requisitos al comienzo del proyecto (entre otras cosas, porque hasta que no vea evolucionar el proyecto no tendrá una idea clara de qué es lo que realmente quiere).
- No habrá disponible una versión operativa del sistema hasta llegar a las etapas

finales del proyecto, por lo que la rectificación cualquier decisión tomada erróneamente en las etapas iniciales del proyecto supondrá un coste adicional significativo, tanto económico como temporal (y eso sin tener en cuenta la mala impresión causada por un retraso en la fecha de entrega).



El ciclo de vida clásico: Modelo "en cascada".

Tal cual, el modelo de ciclo de vida en cascada no nos indica nada acerca de la relación contractual existente entre el cliente y la organización encargada del desarrollo de software. Desde el punto de vista de una empresa de desarrollo de software, formalizar la firma de un contrato al final de la etapa de análisis, por ejemplo, puede ayudar a reducir el riesgo que supone elaborar un presupuesto cuando aún no se dispone de toda la información necesaria para que la estimación del esfuerzo requerido por el proyecto sea lo suficientemente precisa. Este tipo de contrato obliga a que el cliente se haga cargo de los costes adicionales ocasionados por cambios en los requerimientos, mientras que la empresa de desarrollo de software deberá asumir los gastos ocasionados si el producto finalmente entregado no cumple todas las condiciones pactadas a la firma del contrato.

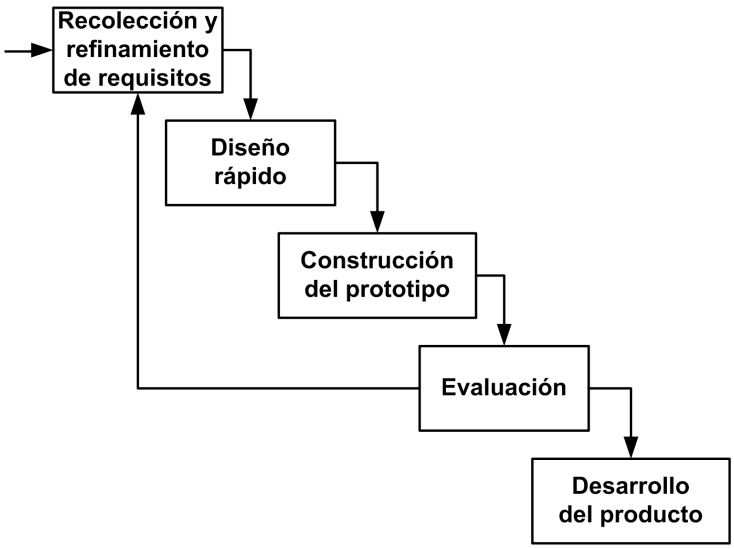
Por desgracia, un modelo contractual como el descrito en el párrafo anterior no siempre resulta aceptable para el cliente, que puede verse obligado a invertir dinero a cambio de nada. Esto podría pasar si, tras la etapa de análisis, el proyecto se desestima por no ser técnica o

económicamente viable. Es más, si el cliente acepta a regañadientes la firma de un contrato al final de la etapa de análisis, la imagen de la empresa desarrolladora de software puede verse seriamente deteriorada en cuanto surja cualquier tipo de problema.

Para limar las asperezas que pueden surgir en la relación cliente-proveedor y mejorar el rendimiento del equipo del proyecto, hoy en día se suele recurrir a modelos iterativos como los que se describirán a continuación.

Desarrollo de prototipos

Normalmente, el cliente es capaz de definir un conjunto general de objetivos para el sistema que hemos de construir, pero no identifica los requisitos detallados. En otros casos, puede que nosotros no estemos seguros de la eficiencia de un algoritmo, de la capacidad de nuestro diseño para soportar los requerimientos del sistema o de la forma en que debe diseñarse la interfaz de usuario. En cualquiera de estas situaciones, resulta adecuado construir un prototipo.



Prototipado

El desarrollo de prototipos reduce el riesgo de que nuestro proyecto fracase y facilita la especificación de requerimientos de productos que desconocemos. Sin embargo, también tiene sus inconvenientes: el cliente puede pensar que el prototipo es el sistema definitivo, ignorando que un prototipo no es un sistema acabado aunque tenga el mismo aspecto externo. Esto puede conducir a la consolidación de aspectos de baja calidad de un prototipo en el

sistema final que se entrega si el prototipo no se desecha a tiempo.

Fred Brooks nos aclara lo que hay que hacer cuando un prototipo ya ha cumplido con su propósito: *"En la mayoría de los proyectos, el primer sistema que se construye apenas resulta utilizable. Puede que sea demasiado lento, demasiado grande, difícil de usar o las tres cosas a la vez. No queda más remedio que comenzar de nuevo y construir una versión rediseñada que resuelva los problemas... Cuando se utiliza un concepto nuevo... hay que construir un sistema para desecharlo, porque incluso la mejor planificación no puede asegurar que vaya a salir bien la primera vez. Por tanto, la cuestión no es si hay que construir un sistema piloto y desecharlo. Se desechará. La única cuestión es si planificar de antemano la construcción de algo que se va a desechar, o prometer la entrega del desecho a los clientes..."* (*The Mythical Man-Month*, "El mítico hombre-mes", 1975, uno de los libros de gestión de proyectos de desarrollo de software más populares que jamás se han escrito).

A veces, los prototipos desechables no se llegan a desechar. Pero los prototipos no siempre son desechables. En tal caso, estaremos utilizando un modelo iterativo de refinamiento de prototipos en el que, tras varias iteraciones, seremos capaces de construir un sistema que se adapte mejor a las necesidades de nuestro cliente.

Modelos iterativos

En 1994, el Departamento de Defensa de Estados Unidos (el mayor contratista mundial de proyectos de desarrollo de software) cambió oficialmente sus estándares de desarrollo de software y descartó el modelo en cascada para introducir el estándar 498, que utiliza un modelo iterativo de desarrollo de software.

Los modelos iterativos consisten en descomponer un proyecto de desarrollo de software en una serie de subproyectos de menor envergadura. Estos subproyectos deben diseñarse de tal forma que cada uno de ellos aporte funcionalidad nueva para el sistema desde el punto de vista del usuario final del mismo.

Usualmente, las personas involucradas en el proyecto establecen prioridades entre los requerimientos iniciales del sistema para decidir qué parte del mismo se construirá primero. El cliente y los usuario finales abogarán por darle prioridad a las funciones más útiles del sistema (o las más "vendibles"). Por otro lado, los diseñadores del sistema deberán determinar las dependencias existentes entre sus distintos componentes y priorizar aquéllos que supongan un riesgo mayor para la viabilidad final del proyecto. Las prioridades de unos y otros habrán de consensuarse razonablemente y servirán para determinar el ámbito de los subproyectos en que se descompondrá el proyecto inicial.

Los modelos iterativos de desarrollo de software permiten adelantar el momento en el que se determina si un proyecto es técnicamente viable o no (con lo que se eliminan costes innecesarios si, finalmente, el proyecto hubiese de cancelarse). También promueven una mejor comunicación con el usuario/cliente, ya que se dispondrá antes de una versión operativa

del sistema, aunque sea de funcionalidad reducida. Estas versiones intermedias del producto ayudan a la eliminación de malentendidos que pueden surgir en la etapa de elicitación de requerimientos. Además, ayudan a que el usuario se forme una idea más clara de lo que realmente necesita.

¿Secuencial o iterativo?

El modelo de desarrollo más adecuado para un proyecto dependerá del tipo de sistema que se ha de construir:

- En general, se elegirá un modelo secuencial cuando los requerimientos se conocen bastante bien y son estables, cuando el diseño será similar al de otros sistemas con los que tenemos experiencia, cuando los integrantes del equipo de desarrollo ya se conocen y están familiarizado con el entorno de desarrollo, o cuando el coste de tener que cambiar algo en las etapas finales del proyecto resultaría prohibitivo.

- En la práctica, no obstante, los modelos iterativos se adaptan mejor a la realidad del desarrollo de software (especialmente en sistemas medianos y grandes). Nos decantaremos por un modelo iterativo cuando los requerimientos no se conocen con exactitud o se prevé que puedan cambiar en el futuro, cuando el diseño del sistema es complejo o no tiene precedentes para nosotros, cuando el proyecto en sí es arriesgado económicamente y cuando podamos controlar el coste de futuros cambios en el sistema (algo que siempre tendremos que hacer si tenemos en cuenta lo que aprendimos al estudiar la etapa de mantenimiento).

Al seguir un modelo iterativo, puede que le dediquemos muy poco tiempo a las etapas iniciales del ciclo de vida de un sistema, lo que puede causar una tasa de cambios tan alta que impida que el proyecto progrese. Del mismo modo, si les dedicamos demasiado tiempo, hasta el punto de seguir a pies juntillas los requerimientos iniciales del sistema, puede que estemos negando la realidad con la que nos encontramos y, de nuevo, impidamos que el proyecto progrese adecuadamente.

Para planificar un proyecto que siga un modelo iterativo, primero se prepara una descomposición a grandes rasgos del proyecto en una serie de iteraciones, cada una de las cuales se considerará como un proyecto independiente. En vez de realizar una planificación

detallada de todo el proyecto, sólo se detallará el plan correspondiente a la primera iteración. Sí se establecerán las fechas de inicio y finalización de las distintas iteraciones y se definirán los objetivos principales de cada una de ellas. Llegado el caso, estos objetivos se pueden redefinir conforme avance el proyecto.

El modelo en espiral de Boehm

A lo largo de los años se han propuesto multitud de modelos iterativos de desarrollo de software. A continuación se describen algunos de los más conocidos:

- El **modelo en espiral** de Barry Boehm hace especial hincapié en la prevención de riesgos. Este modelo define cuatro actividades principales: planificación (determinar los objetivos, alternativas y restricciones del proyecto), análisis de riesgos (análisis de alternativas e identificación/resolución de riesgos), ingeniería (desarrollo del producto) y evaluación (revisión por parte del cliente y valoración de los resultados obtenidos de cara a la siguiente iteración). En cada iteración alrededor de la espiral se construyen versiones cada vez más completas del software.

- Los **modelos evolutivos** (como el **modelo Evo** de Tom Gilb o los **modelos ágiles** populares hoy en día, entre los que se encuentra la auto-denominada **programación extrema**) se caracterizan por realizar entregas por etapas del sistema. Usualmente, el proyecto se descompone en iteraciones de longitud fija (de 1 a 6 semanas) y cada iteración ha de proporcionar algún aspecto completo de la funcionalidad del sistema. Cada ciclo se concentra en las funciones de mayor valor añadido. De esta forma, si se cancelase el proyecto en cualquier momento, el usuario siempre tendrá lo máximo que se puede conseguir con los recursos

invertidos hasta el momento. Igualmente, se puede prorrogar el proyecto si se considera interesante seguir añadiéndole funcionalidad al proyecto.

- También existen otros modelos, conocidos por el nombre de **modelos de estabilización y sincronización**, en los que se sigue la misma estrategia que en los modelos iterativos pero sin llegar a realiza una entrega por etapas del sistema. Éste es el caso, por ejemplo, del modelo de desarrollo de software utilizado internamente en empresas como Microsoft.

Marcos para el proceso de desarrollo de software

Como hemos visto, existe una amplia variedad de propuestas en lo que respecta a cómo organizar el proceso de desarrollo de software. La mayoría de las propuestas son prescriptivas (definen qué actividades hay que realizar y en qué orden), si bien algunas propuestas van más allá y definen marcos para organizar el conjunto de actividades y tareas involucradas en un proyecto de desarrollo de software. Estos marcos sugieren qué combinaciones de actividades son las más indicadas en cada etapa del proceso de desarrollo de software y cuáles deberían ser los resultados que se obtengan de cada una de ellas. Los siguientes son algunos de los más populares hoy en día:

- El modelo **CMMI** (*Capability Maturity Model - Integrated*, propuesto por el Instituto de Ingeniería del Software de la Universidad de Carnegie-Mellon) identifica una serie de áreas clave en términos de objetivos específicos y prácticas necesarias para lograr esos objetivos. Los objetivos establecen las características que el proceso de desarrollo de software debe tener para que las actividades de cada área puedan ser efectivas. En cierto sentido, CMMI viene a ser como una certificación de calidad ISO 9000 adaptada a proyectos de desarrollo de software. Una certificación de este tipo puede ayudar a conseguir determinados tipos de proyectos (gubernamentales, principalmente), si bien sólo garantiza que el proyecto se realiza siguiendo un proceso definido, no que las distintas tareas se realicen correctamente. De hecho, el proceso puede estar perfectamente definido y "certificado" sin ser el proceso adecuado para el proyecto que tengamos entre manos.

- El Proceso Unificado de Rational (**RUP**, *Rational Unified Process*, propuesto originalmente por una empresa llamada Rational Software Corporation que hoy es una división de IBM) describe un marco adaptable para procesos iterativos de desarrollo de software. El ciclo de vida de un proyecto RUP se divide en ciclos de desarrollo individuales que, a su vez, se descomponen en cuatro fases principales (incepción, elaboración, construcción y transición). Para cada una de estas fases, RUP identifica qué disciplinas (actividades) son las más relevantes. Finalmente, en un proyecto concreto cada fase viene definida por una serie de objetivos y se descompone en iteraciones de duración fija (de, por ejemplo, 3 semanas).

El ciclo de vida de una base de datos

Una base de datos no es más que un componente de un sistema de información. Por tanto, el ciclo de vida del sistema de información incluye el ciclo de vida de la base de datos que forma parte de él. En particular, desde el punto de vista de la base de datos, centraremos principalmente nuestra atención en las siguientes actividades:

- **Definición del sistema**: Durante la etapa de análisis de requerimientos del sistema, nos fijaremos especialmente en todos los requerimientos asociados a los datos con los que ha de trabajar nuestro sistema.

- **Diseño de la base de datos**: El análisis de los requerimientos del sistema nos permitirá organizar los datos con los que nuestro sistema habrá de trabajar. Este proceso de diseño, que está íntimamente ligado a la futura base de datos de nuestro sistema, lo descompondremos en tres fases:

  · **Diseño conceptual** (descripción del esquema de la base de datos utilizando un modelo de datos conceptual).

  · **Diseño lógico** (descripción de la base de datos con un modelo de datos implementable, como puede ser el caso del modelo relacional).

  · **Diseño físico** (descripción de la base de datos a nivel interno, de acuerdo con las características del sistema gestor de bases de datos que decidamos utilizar).

- **Implementación de la base de datos** (la parte de la implementación del sistema correspondiente a la creación de la base de datos).

- **Carga o conversión de los datos**: Como parte de la instalación o despliegue del sistema, tendremos que introducir en la base de datos todos aquellos datos que resulten necesarios para que las aplicaciones de nuestro sistema de información puedan funcionar. Como parte de esta *inicialización* de la base de datos, puede que resulte necesario extraer datos de otro sistema y convertirlos a un formato adecuado para nuestro sistema (entre otras cosas, porque el esquema de nuestra base de datos probablemente diferirá del esquema de las bases de datos de las que se extraigan los datos necesarios para arrancar nuestro sistema).

- **Conversión de aplicaciones**: Si determinadas aplicaciones (que ya existiesen anteriormente al diseño de nuestro sistema) han de seguir funcionando, dichas aplicaciones deberán adaptarse al esquema de nuestra base de datos. Por tanto, como parte del mantenimiento de dichas aplicaciones, tendremos que diseñar los mecanismos adecuados para que estas aplicaciones puedan seguir funcionando

correctamente sobre una base de datos diferente a la base de datos sobre la que fueron diseñadas inicialmente. A veces, podremos solucionar este problema creando vistas adecuadas de nuestra base de datos para tales aplicaciones. Otras veces, tendremos que modificar la implementación de las aplicaciones antiguas e, incluso, rehacerlas casi por completo.

- **Verificación y validación**: Como en todo sistema de información, deberemos verificar que la base de datos y las aplicaciones funcionan correctamente. Además, deberemos comprobar que el sistema construido se ajusta a las necesidades reales que promovieron su proyecto de desarrollo (esto es, validar el sistema y sus requerimientos).

- **Operación, supervisión y mantenimiento**: Finalmente, una vez puesto en marcha el sistema, se llega a la etapa "final" del ciclo de vida de todo sistema de información (en la que, como ya vimos, se repetirá todo el ciclo cada vez que tengamos que realizar modificaciones sobre el sistema ya existente).

De las actividades descritas en los párrafos anteriores, todas ellas relacionadas directamente con la base o bases de datos utilizadas en un sistema de información, estudiaremos a fondo las correspondientes a las etapas iniciales del ciclo de vida de la base de datos. Antes de estudiar técnicas concretas, no obstante, detallares algo más el proceso de diseño que utilizaremos para construir correctamente una base de datos.

El proceso de diseño de una base de datos

El problema de diseñar bases de datos consiste en diseñar la estructura lógica y física de una o más bases de datos para atender las necesidades de información de los usuarios de un conjunto definido de aplicaciones. Estos usuarios pueden pertenecer todos a una organización concreta (como sucede con los trabajadores de una empresa o los funcionarios de un organismo público), o bien formar parte de un colectivo con intereses comunes (tal es el caso de los usuarios de multitud de aplicaciones web, desde un buscador como Google hasta un servicio de información geográfica tipo Páginas Amarillas).

Antes de pasar a ver la metodología que utilizaremos para diseñar bases de datos, hay que recordar que el diseño de bases de datos es sólo una de los procesos involucrados en la construcción de un sistema de información. Generalmente, para construir un sistema de información se llevarán a cabo distintas actividades paralelas:

- Por un lado, será necesario diseñar el contenido y la estructura de la base de datos que dará soporte al sistema de información.

- Por otro, también será imprescindible diseñar el conjunto de aplicaciones que le permitirán al usuario sacar partido del sistema de información.

Tanto en las actividades relacionadas con los datos del sistema (todo lo relativo a la base de datos) como en aquéllas relacionadas con los procesos del mundo real que el sistema trata de mejorar (mediante un conjunto de aplicaciones), resulta recomendable el uso de una metodología apropiada.

En esencia, la metodología utilizada en un proyecto no es más que el conjunto de convenciones que los integrantes de un equipo de trabajo acuerden emplear. Esta definición incluiría, por ejemplo, a la metodología aSdM utilizada por algunas empresas de desarrollo de software (una referencia irónica al hecho de ir haciendo las cosas "a salto de mata"). Sin embargo, por metodología usualmente se entiende algo más. Si acudimos a un diccionario, encontraremos que una metodología es un conjunto de métodos (sic), aplicados de forma sistemática.

Una buena metodología de diseño ha de incluir todo lo que normalmente resulte necesario para obtener un buen diseño. Generalmente, una metodología, que implicará el uso de métodos y técnicas adecuadas a nuestro problema, se centrará en la coordinación de las actividades que han de realizarse. De acuerdo con las etapas del ciclo de vida de un sistema de información, una metodología de diseño descompone el proceso de diseño en una serie de etapas. Para cada una de las etapas, propondrá el uso de determinadas técnicas y herramientas de diseño, así como la generación de una serie de documentos que facilitarán la transición de una etapa a la siguiente.

A continuación, presentaremos las distintas fases en las que descompondremos el proceso de diseño de bases de datos. Para cada una de las fases, mencionaremos sus objetivos concretos, las técnicas particulares que recomendamos utilizar en cada etapa y los documentos que se deberían obtener como resultado de cada una de ellas.

FASES DEL DISEÑO DE BASES DE DATOS

- Análisis de requisitos
- Diseño conceptual
- Elección del sistema gestor de bases de datos
- Diseño lógico
- Diseño físico
- Instalación y mantenimiento

Fase 1: Análisis de requerimientos

Objetivo

Recabar información sobre el uso que se le piensa dar a la base de datos.

Tareas

Elicitación de los requisitos del sistema:

- Identificación de las principales áreas de la aplicación y grupos de usuarios.
- Estudio y análisis de la documentación existente relativa a las aplicaciones.
- Estudio del entorno de operación actual.
- Estudio del uso de la información (transacciones, frecuencias y flujos de datos).

Resultados

Documento de especificación de requerimientos:

- Descripción del sistema en lenguaje natural.
- Lista de requerimientos (organizados de forma jerárquica).
- Diagramas de flujo de datos (DFD).
- Casos de uso.

Fase 2: Diseño conceptual

Objetivo

Producir un esquema conceptual de la base de datos (independiente del sistema gestor de bases de datos que luego vayamos a utilizar).

Tareas

- Comprensión de la estructura, semántica, relaciones y restricciones asociados a los datos que deben almacenarse en la base de datos.
- Modelado de los datos del sistema (obtención de una descripción estable de lo que será el contenido de la base de datos).
- Comunicación entre usuarios finales, analistas y diseñadores para comprobar la validez del modelo obtenido.

Resultados

- Diagrama E/R, diagrama CASE*Method o diagrama de clases UML.
- Diccionario de metadatos.

Fase 3: Elección del SGBD

La elección del sistema gestor de bases de datos que vayamos a utilizar se realiza en dos etapas:

- Primero se realiza la **elección del modelo de datos**, el tipo de sistema gestor de bases de datos que vamos a usar: relacional, objeto-relacional, orientado a objetos, multidimensional...

- A continuación se elige el sistema gestor de bases de datos concreto (y su versión). Por ejemplo, si decidimos utilizar un sistema gestor de bases de datos relacionales, podemos recurrir al gestor de bases de datos de Oracle, al DB2 de IBM, al SQL Server de Microsoft, al Interbase de Borland o a cualquier otro de los muchos sistemas gestores de bases de datos relacionales que existen en el mercado.

Selección de un sistema gestor de bases de datos

Un sistema gestor de bases de datos (SGBD o DBMS si nos atenemos a sus siglas en inglés) es un producto software con capacidad para definir, mantener y utilizar bases de datos. El sistema de gestión de bases de datos que decidamos utilizar debe permitirnos, entre otras cosas, definir estructuras de almacenamiento adecuadas y acceder a los datos de forma eficiente y segura. A continuación se enumeran algunos de los aspectos en que deberíamos fijarnos para elegir un SGBD concreto:

Factores técnicos

- Organización de los datos independientemente de las aplicaciones que los vayan a usar (independencia lógica) y de los ficheros en los que vayan a almacenarse (independencia física).

- Datos y aplicaciones accesibles a los usuarios y a otras aplicaciones de la manera más amigable posible (mediante lenguajes de consulta como SQL o Query-by-example).

- Datos gestionados de forma centralizada e independiente de las aplicaciones.

- No redundancia (los datos no deben estar duplicados), consistencia e integridad.

- Fiabilidad (protección frente a fallos en el hardware).

- Seguridad (no todos los datos deben ser accesibles a todos los usuarios y el SGBD debe ayudarnos a controlar esto).

Selección de un sistema gestor de bases de datos

- Capacidad de replicación y distribución.
- Disponibilidad de herramientas adecuadas de desarrollo de software.
- Portabilidad.

Factores no técnicos

- Coste de la adquisición del software (licencias de uso del SGBD).
- Coste del hardware necesario para el uso del SGBD.
- Costes asociados al mantenimiento de la base de datos.
- Coste de creación y conversión de la base de datos.
- Coste de personal (tanto de formación como de operación de la base de datos).
- Disponibilidad de servicios por parte del proveedor del SGBD.

Fase 4: Diseño lógico

Objetivo

Crear el esquema conceptual de la base de datos (y todos los esquemas externos asociados a las distintas aplicaciones del sistema) de acuerdo con el modelo de datos del sistema gestor de base de datos elegido.

Tareas

Para realizar el diseño lógico de la base de datos, hay que transformar los esquemas obtenidos en el diseño conceptual en un conjunto de estructuras propias del modelo abstracto de datos elegido. En el caso de las bases de datos relacionales tendremos que realizar las siguientes tareas:

- Paso del diagrama E/R (o equivalente) a un conjunto de tablas.
- Normalización de las tablas

Resultado

Un conjunto de estructuras propias del modelo abstracto de datos del SGBD elegido (esto es, un conjunto de tablas si trabajamos con bases de datos relacionales).

Fase 5: Diseño físico

Objetivo

El diseño físico de la base de datos consiste en elegir las estructuras de almacenamiento apropiadas (tablas, particiones de tablas, índices...) para que el rendimiento de la base de datos sea el adecuado para las distintas aplicaciones a las que ha de dar servicio.

Sobre el rendimiento de la base de datos

Por rendimiento de las aplicaciones se entiende el tiempo de respuesta del sistema a las peticiones del usuario, el aprovechamiento del espacio de almacenamiento en disco utilizado por la base de datos, la productividad de las transacciones de la base de datos y cualquier otro aspecto que pueda afectar a la percepción del sistema por parte del usuario.

Usualmente, el rendimiento del sistema dependerá del tamaño de la base de datos, del número de registros de cada tipo que ha de almacenar y del número de usuarios que accederán concurrentemente a la base de datos, así como de los patrones concretos de inserción, actualización y obtención de datos.

Tareas

- Estimar adecuadamente los diferentes parámetros físicos de nuestra base de datos, para lo cual podemos recurrir a técnicas analíticas (modelos matemáticos del rendimiento de un sistema) y a técnicas experimentales (como la construcción de prototipos, el uso de técnicas de simulación o la realización de pruebas de carga).

- Preparar las sentencias DDL correspondientes a las estructuras identificadas durante la etapa de diseño lógico de la base de datos.

Resultado

Un conjunto de sentencias DDL escritas en el lenguaje del SGBD elegido (incluyendo la creación de índices, la selección de parámetros físicos de la base de datos, etcétera).

Fase 6: Instalación y mantenimiento

Como en cualquier sistema de información, casi siempre resulta necesario modificar el diseño de la base de datos, ya sea porque el rendimiento observado después de la implementación del sistema de bases de datos no sea el adecuado o porque haya que introducir cambios en el esquema de la base de datos como consecuencia del mantenimiento del sistema de información. Por ambos motivos se incluye explícitamente esta fase en el proceso de diseño de bases de datos.

Instalación y puesta en marcha

- La instalación de la base de datos suele ser responsabilidad del administrador de la base de datos (DBA: *Database Administrator*), que se encarga de recopilar todas las sentencias DDL necesarias para crear los distintos esquemas de la base de datos.

- Una vez creados estos esquemas, se procede a la carga inicial de los datos en la base de datos, para lo cual puede ser necesaria la implementación de rutinas de conversión, tal como vimos al describir el ciclo de vida de una base de datos.

Mantenimiento

- Casi todos los sistemas gestores de bases de datos incluyen alguna utilidad que nos permite supervisar el funcionamiento del sistema. Dichas utilidades de monitorización recopilan información estadística del uso del sistema para su análisis posterior, lo que nos facilitará todas las tareas relacionadas con la optimización del rendimiento del sistema.

- Cuando los requisitos del sistema cambien y haya que actualizar las aplicaciones de nuestro sistema de información, el esquema de la base de datos también se verá sometido a algunas modificaciones.

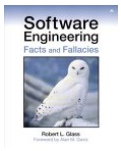
Cuando se detecta un rendimiento deficiente del sistema, no siempre es suficiente con ajustar los parámetros de configuración del sistema gestor de bases de datos.

En ocasiones, basta con la reorganización de las estructuras internas de la base de datos (por ejemplo, mediante la creación de los índices adecuados) pero también hay veces en que resulta necesaria la creación de tablas redundantes (vistas materializadas). En estos casos, debemos ser especialmente cuidadosos para asegurar la consistencia de los datos almacenados en la base de datos (algo que, generalmente, se puede conseguir mediante la implementación de disparadores o *triggers* en el propio gestor de bases de datos).

Bibliografía

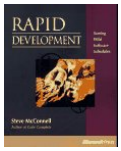
Ramez A. Elmasri & Shamkant B. Navathe: *"Fundamentos de Sistemas de Bases de Datos"*, Addison-Wesley, 2002 [3ª edición]. ISBN 8478290516

En este libro se puede encontrar una buena descripción del proceso de diseño de bases de datos, tal como lo acabamos de contar en la sección anterior.



Robert L. Glass: *"Facts and Fallacies of Software Engineering"*, Addison-Wesley, 2003. ISBN 0321117425

El título original de este excelente libro da una buena pista sobre lo que nos podemos encontrar al abrirlo: *Fifty-Five Frequently Forgotten Facts (and a Few Fallacies) about Software Engineering* [55 hechos comúnmente olvidados (y unas cuantas falacias) sobre Ingeniería del Software]. Al editor pareció no gustarle el título original para el libro "F" y éste fue modificado. Esperemos que no fuese porque su base de datos no admitía un título tan largo...



Steve McConnell: *"Rapid Development: Taming wild software schedules"*, Microsoft Press, 1996. ISBN 1556159005

Este libro, bastante más ameno e informativo que la mayor parte de libros de texto de Ingeniería del Software (aunque también algo más sesgado que ellos, a decir verdad), incluye información relativa a una amplia variedad de técnicas útiles en las distintas etapas de un proyecto de desarrollo de software. En particular, se puede encontrar en él gran cantidad de información relativa a la fase de la planificación de un proyecto y, también, a la gestión de un proyecto a lo largo de todo su ciclo de vida.

Bibliografía complementaria

Los siguientes libros resultan altamente recomendables para aprender más acerca de las fases del ciclo de vida de un sistema de información que aquí apenas se han mencionado:

Martin Fowler: *"Patterns of Enterprise Application Architecture"*, Addison-Wesley, 2003. ISBN 0321127420

> Este libro está dedicado íntegramente al diseño de las aplicaciones que se utilizan en la gran mayoría de los sistemas de información de cualquier empresa. En particular, describe cómo crear arquitecturas multicapa utilizando técnicas de diseño orientado a objetos e incluye un catálogo de patrones de diseño extremadamente útil para cualquier diseñador de aplicaciones de gestión.

Steve McConnell: *"Code Complete: A practical handbook of software construction"*, Microsoft Press, 2ª edición, 2004. ISBN 0735619670

> Si el libro de Fowler cubre aspectos de diseño de aplicaciones, el de McConnell no se queda corto a la hora de analizar todo lo relacionado con la construcción de software, desde el diseño detallado de módulos de un programa hasta el uso de técnicas de optimización de código, sin olvidar (casi) ningún detalle relativo al uso de variables o estructuras de control en un programa.

Cem Kaner, James Bach & Bret Pettichord: *"Lessons learned in software testing"*, Wiley Computer Publishing, 2002. ISBN 0471081124

> La etapa de pruebas suele estar peor vista que las de análisis, diseño e implementación, por lo que resulta algo más difícil encontrar buenos libros sobre el tema (entre otras cosas, por ser el destino inicial de muchos al ser contratados por una empresa, tras lo cual suelen ascienden a programadores, de programadores a analistas, y de analistas a jefes de proyecto). Este libro, no obstante, es bastante bueno y tiene un estilo similar al libro "F" de Robert Glass. Además, incluye todas las referencias necesarias para el que desee profundizar más en el tema.

PD

La etapa de mantenimiento esta aún peor vista que la de pruebas y, acerca de ella, es prácticamente imposible encontrar una referencia en condiciones que cubra el tema con exhaustividad. Sí existen, no obstante, buenas monografías dedicadas a temas específicos relacionados con el mantenimiento. Éste es el caso de *Working Effectively with Legacy Code* de Michael Feathers (Prentice Hall PTR, 2004, ISBN 0131177052).