Diseño de Sistemas

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 y ocultar sus detalles** de implementación. 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. 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.

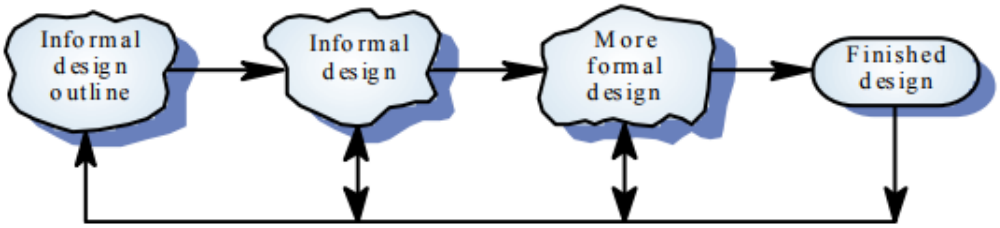
El diseño de un sistema de información también presenta distintas facetas:

- 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.

El **Diseño de Software** es la acción de construir soluciones que satisfagan los requerimientos del cliente. Existen varias etapas en el proceso de diseño de software, a saber son:

- Entendimiento del problema
- Identificar una o más soluciones
- Describir abstracciones de la solución
- Repetir el proceso para cada abstracción identificada hasta que el diseño esté expresado en términos sencillos

Cualquier diseño debe ser modelado como una gráfica dirigida hecha de entidades con atributos los cuales participan en relaciones. El sistema debe estar descrito a distintos niveles de abstracción y el diseño ocurre en etapas que se traslapan. La primera idea que se tiene al construir una solución de un determinado problema es un modelo mental que constituye el primer intento de diseño llamado comúnmente diseño informal. Este diseño a medida que se va describiendo en papel utilizando técnicas y procedimientos esquemáticos y metódicos va adquiriendo forma hasta constituirse en un diseño formal equivalente. La siguiente figura ejemplifica este hecho:



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. 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.

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

En las arquitecturas 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 estar especialmente diseñada para agilizar su trabajo. Además, suele ser recomendable diferenciar lo que se suministra (presentación) de lo que se consume (acceso a los servicios suministrados por otros sistemas).

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 encapsulan los conjuntos de datos con los que se trabaja y la lógica asociada a ellos.

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 capa de servicio 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).

Formas de organizar el código

Excepto algunos scripts muy simples, cualquier aplicación acaba constando de muchos archivos que necesitamos organizar de alguna manera en carpetas para poder manejarlo.

Existen muchas formas de organizar el código y en ellas influye mucho el gusto de cada uno o las convenciones aplicadas por cada equipo de desarrollo, pero eso no quita que también haya formas más o menos estándar, cada una de ellas centrada en mejorar determinados aspectos.

Organización por roles tecnológicos

La organización por roles es una de las más extendidas actualmente en el desarrollo de software. Probablemente uno de los proyectos que más hizo por este tipo de organización fue Ruby on Rails con su convención sobre configuración basada en la ubicación física en disco de sus modelos, vistas y controladores, y que luego ha sido copiada por otros frameworks.

Esta organización se basa en agrupar los archivos en base al rol tecnológico que desempeña en el sistema. De esta forma, si en la capa de presentación estamos aplicando un patrón MVC, tendremos carpetas con Models, Views y Controllers.

La organización por roles tecnológicos fomenta una forma de pensar homogénea y centrada en aspectos técnicos. Al definir tanto los roles y hacerlos tan explícitos con las carpetas independientes, sirve para remarcar mucho la arquitectura del sistema y eso nos lleva a encajar las nuevas funcionalidades dentro de esa arquitectura.

La principal ventaja de esto es que el diseño del sistema tiende a mantenerse muy homogéneo y la estructura de carpetas actúa como guía al implementar nuevas funcionalidades.

Es una buena opción cuando se está trabajando con equipos de desarrollo grandes en los que hay más rotación de personal, debido a que determina pautas muy claras de **cómo hacer las cosas** y el resultado final resulta más predecible.

Precisamente esta homogeneización se puede convertir en uno de los inconvenientes de esta forma de organizar el código, porque lleva a intentar encajar todas las funcionalidades del sistema en una serie de roles que, a veces, no son necesarios o no son necesariamente la mejor forma de implementar una funcionalidad concreta.

Organización por funcionalidades

La organización por funcionalidades (features) se basa en primar el aspecto funcional sobre el técnico a la hora de decidir dónde colocar cada archivo.

En lugar de agrupar los ficheros por el rol que tienen (Controllers, Repository, Views, etc), los agrupamos en base a la funcionalidad que implementan.

Por ejemplo, en una aplicación web MVC que tiene una parte para gestionar clientes podríamos tener una carpeta clientes que incluye el modelo, la vista, el controlador y el resto de cosas que necesitemos.

Llevado al extremo, en cada carpeta tendríamos una parte de la aplicación, desde el UI hasta la persistencia, pero normalmente no se llega a tanto, y podemos tener una carpeta clientes con el código javascript y html de nuestra aplicación hecha con Angular, y en alguna otra parte otra carpeta clientes con el código en python de un proyecto WebAPI, con su controlador, servicios para cargar datos, etc.

Si la organización basada en roles tenía cierto parecido con las arquitecturas N capas, podríamos decir que la organización basada en funcionalidades se asemeja más (salvando las distancias) al concepto de microservicios, porque fomenta el minimizar las dependencias entre unas carpetas y otras, para tener funcionalidades lo más cohesivas e independientes posibles.

Esta forma de organizar el código evita los inconvenientes de la organización por roles. Con ella todo el código relativo a una funcionalidad está físicamente cerca, ayudando a saber mejor qué código está ligado a esa funcionalidad y a navegar por él.

Además facilita la navegación por la aplicación a un nivel más alto. Cuando ves carpetas llamadas Clientes, Pedidos o Catálogos, es más fácil hacerse a la idea de qué hace la aplicación que si ves carpetas llamadas Controllers, Views y Models.

A la hora de implementar nuevas funcionalidades ayuda a considerarlas de forma independiente, por lo que resulta más sencillo «separarse» del diseño general y utilizar diseños más optimizados para cada funcionalidad concreta sin necesidad de ajustarnos a unos roles marcados previamente.

Esa libertad para implementar nuevas funcionalidades se convierte en un arma de doble filo si estás trabajando con gente que no es capaz de diseñar con un mínimo de sentido común, puesto que resulta más complicado fomentar el uso de buenas prácticas estandarizadas de diseño.

Mezclando ambas alternativas

Hasta ahora hemos visto dos formas de organizar el código que parten de ideas distintas y priorizan objetivos diferentes, pero no hace falta ser muy listo para darse cuenta de que en realidad estamos hablando de dos ejes distintos, roles técnicos y funcionalidades, y que en el fondo son dos conceptos ortogonales, por lo que podemos mezclarlos. Por lo normal cuando una aplicación se hace realmente compleja se termina mezclando de una forma u otra.

Si se ha empezado organizando por roles técnicos, llega un momento en que tener muchos “controllers” en una sola carpeta se hace inmanejable y se acaba separándolos en distintas carpetas dentro de la carpeta.

Igualmente, si cada funcionalidad de una aplicación acaba teniendo muchos filtros, controllers y vistas, es razonable que terminemos particionando más los archivos, ya sea mediante carpetas o alguna convención de nombres en los propios archivos.