# Definición y características

En la fase de diseño, tomando como punto de partida los requisitos (funcionales y no funcionales) se pretende obtener una descripción de la mejor solución software/hardware que dé soporte a dichos requisitos, teniendo en cuenta no solamente aspectos técnicos, sino también aspectos de calidad, coste y plazos de desarrollo. Es importante resaltar que en esta fase se pasa del *qué* (obtenido en la fase de requisitos) al *cómo* (que es el objetivo del diseño).

En la fase de diseño se pasa a un ámbito más técnico donde se habla, fundamentalmente, el lenguaje de los desarrolladores, ya que los productos del diseño son documentos y artefactos orientados a ingenieros del software, y poco o nada el de los clientes, pues la comunicación con éstos o con los usuarios es limitada, si bien puede ser necesario generar documentación de diseño para los clientes llegado el caso. Con todo lo anterior en mente, es el momento de proporcionar una definición formal del diseño. Concretamente, el Glosario IEEE de Términos de Ingeniería del Software (IEEE, 1990) lo define con las dos acepciones siguientes:

> El *diseño* puede definirse como (1) el proceso para definir la arquitectura, los componentes, las interfaces, y otras características de un sistema o un componente, y (2) como el resultado de este proceso.

En esta definición se mencionan varios elementos importantes en los diseños del software como por ejemplo, la arquitectura, que tiene que ver con la descomposición de un sistema en sus componentes e interfaces. No obstante, no aclara un hecho importante sobre la estructura del propio proceso de diseño que es esencial y que se refiere a que el diseño se descompone claramente en dos subprocesos:

- *Diseño de la arquitectura (o diseño de alto nivel)*, en el cual se describe cómo descomponer el sistema y organizarlo en los diferentes componentes (lo que se conoce como la arquitectura del software).
- *Diseño detallado*, en el que se describe el comportamiento específico de cada uno de los componentes de software identificados.

En términos muy generales, se puede decir que la arquitectura es la visión de alto nivel que posteriormente se detalla hasta el nivel de componentes, elementos modulares que representan la solución final.

La especificación de los componentes, módulos o fragmentos software del sistema, y del modo en que éstos se comunican, se lleva a cabo sin describir detalles internos. Las comunicaciones se expresan con la especificación de las operaciones que los componentes exponen para que otros puedan usarlos, lo que comúnmente se conoce como *interfaces*.

> Un *componente* es una parte funcional de un sistema que oculta su implementación proporcionando su realización a través de un conjunto de interfaces.

Por lo tanto, un componente es generalmente un elemento reemplazable dentro de una arquitectura bien definida que se comunicará con otros componentes a través de interfaces.

> Una *interfaz* describe la frontera de comunicación entre dos entidades software, definiendo explícitamente el modo en que un componente interacciona con otros.

El diseño es la única forma mediante la que se puede traducir con precisión los requisitos del cliente en un producto o sistema acabado. El diseño de software es la base de todas las fases posteriores del desarrollo y, fundamentalmente, de la fase de prueba (ver transparencia 7 de tema 3.1).

Sin diseño, existe el riesgo de construir un sistema inestable, un sistema que falle cuando se realicen pequeños cambios, un sistema que sea difícil de probar, un sistema cuya calidad no pueda evaluarse hasta más adelante cuando quede poco tiempo y se haya gastado mucho dinero.

# Principios del diseño software

A lo largo de la historia de la ingeniería del software, los conceptos fundamentales del diseño han ido evolucionando. Aunque con el paso de los años ha variado el grado de interés en cada concepto, todos han soportado la prueba del tiempo. Cada uno proporciona al diseñador de software el fundamento desde el que pueden aplicarse métodos de diseño sofisticados. Todos ayudan a responder las preguntas siguientes:

- ¿Qué criterios se usan para dividir el software en sus componentes individuales?
- ¿Cómo se extraen los detalles de la función o estructura de datos a partir de la representación conceptual del software?
- ¿Cuáles son los criterios uniformes que definen la calidad técnica de un diseño de software?

## Abstracción

Como en el resto de los problemas de ingeniería, en el desarrollo de una solución de software, el resultado se representará de forma abstracta con diferentes grados de detalle: Partiendo desde un nivel de abstracción alto, y refinando dicha solución, se llega hasta un nivel de detalle próximo a la implementación. En el desarrollo de un sistema se distinguen tres tipos fundamentales de abstracciones:

- *Abstracción de datos*. Define un objeto compuesto por un conjunto de datos. La abstracción *cliente*, por ejemplo, incluirá todos los datos de un cliente tal y como se entiende en el contexto de la aplicación en desarrollo, que podrían ser *nombre*, *dirección*, *teléfono*, etc.
- *Abstracción de control*. Define un sistema de control de un software sin describir información sobre su funcionamiento interno. Una abstracción de control típica es el *semáforo* para describir la coordinación en el funcionamiento de un sistema operativo.
- *Abstracción procedimental*. Aquella que se refiere a la secuencia de pasos que conforman un proceso determinado, por ejemplo, un algoritmo de ordenación.

Estos tipos de abstracciones se usarán a lo largo de todo el diseño, si bien el nivel de detalle irá aumentando a medida que el diseño avanza. Así, los primeros esquemas o bocetos del diseño contendrán abstracciones de muy alto nivel, en los que se oculten un buen número de detalles que en ese punto no sean necesarios. En los últimos momentos del diseño detallado, los diagramas tendrán un nivel de abstracción mucho menor pues la proximidad a la fase de construcción obliga a proporcionar detalles que permitan comprender sin ambigüedad lo expresado.

## División de problemas y modularidad

### División de problemas

La *división de problemas* es un concepto de diseño que sugiere que cualquier problema complejo puede manejarse con más facilidad si se subdivide en elementos susceptibles de resolverse u optimizarse de manera independiente. Un problema es una característica o comportamiento que se especifica en el modelo de los requisitos para el software. Al

separar un problema en sus piezas más pequeñas y por ello más manejables, se requiere menos esfuerzo y tiempo para resolverlo.

Si para dos problemas $p_1$ y $p_2$, la complejidad que se percibe para $p_1$ es mayor que la percibida para $p_2$, entonces se concluye que el esfuerzo requerido para resolver $p_1$ es mayor que el necesario para resolver $p_2$. En general, este resultado es intuitivamente obvio. Lleva más tiempo resolver un problema difícil (ver transparencia 11)

También se concluye que cuando se combinan dos problemas, en la mayoría de los casos la complejidad que se percibe es mayor que la suma de la complejidad considerada por separado. Esto lleva a la estrategia de divide y vencerás, pues es más fácil resolver un problema complejo si se divide en elementos manejables. Esto tiene implicaciones importantes en relación con la modularidad del software (ver transparencia 11),

### Modularidad

La *modularidad* es la manifestación más común de la división de problemas. El software se divide en componentes con nombres distintos y abordables por separado, en ocasiones llamados módulos, que se integran para satisfacer los requisitos del problema.

Desde el punto de vista de la división de problemas, se podría concluir que si el software se dividiera indefinidamente, el esfuerzo requerido para desarrollarlo ¡sería despreciable por pequeño! Desafortunadamente, hay otras fuerzas que entran en juego y que hacen que esta conclusión sea (tristemente) inválida. El esfuerzo (coste) de desarrollar un módulo individual de software disminuye conforme aumenta el número total de módulos. Dado el mismo conjunto de requisitos, tener más módulos significa tamaños individuales más pequeños. Sin embargo, a medida que se incrementa el número de módulos el esfuerzo (coste) asociado con su integración también aumenta. Estas características llevan a una curva de coste total como la que se muestra en la Transparencia 13. Existe un número *M* de módulos que arrojarían el mínimo coste de desarrollo, pero no se dispone de las herramientas necesarias para predecir *M* con exactitud.

## Ocultamiento de información

El concepto de modularidad lleva a una pregunta fundamental: “¿Cómo descomponer una solución de software para obtener el mejor conjunto de módulos?” El principio de ocultamiento de información sugiere que los módulos se “caractericen por decisiones de diseño que se oculten (cada una) de las demás”. En otras palabras, deben especificarse y diseñarse módulos, de forma que la información (algoritmos y datos) contenida en un módulo sea inaccesible para los que no necesiten de ella.

El uso del ocultamiento de información como criterio de diseño para los sistemas modulares, proporciona los máximos beneficios cuando se requiere hacer modificaciones durante las pruebas, y más adelante, al dar mantenimiento al software. Debido a que la mayoría de los datos y detalles del procedimiento quedan ocultos para otras partes del software, es menos probable que los errores inadvertidos introducidos durante la modificación se propaguen a distintos sitios dentro del software.

## Independencia funcional

El concepto de independencia funcional es resultado directo de la separación de problemas y de los conceptos de abstracción y ocultamiento de información. La independencia funcional se logra desarrollando módulos de manera que cada módulo resuelva un subconjunto específico de requisitos y tenga una interfaz sencilla cuando se

vea desde otras partes de la estructura del programa. Es lógico preguntar por qué es importante la independencia.

El software con módulos independientes, es más fácil de desarrollar porque su función se subdivide y las interfaces se siplifican. Los módulos independientes son fáciles de mantener (y probar) debido a que los efectos secundarios causados por el diseño o por la modificación del código son limitados, se reduce la propagación del error y es posible obtener módulos reutilizables. En resumen, la independencia funcional es una clave para el buen diseño y éste es la clave de la calidad del software.

La independencia se evalúa con el uso de dos criterios cualitativos: la cohesión y el acoplamiento. La *cohesión* es un indicador de la fortaleza relativa funcional de un módulo. El *acoplamiento* lo es de la independencia relativa entre módulos.

La cohesión es una extensión natural del concepto de ocultamiento de información. Un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa. Aunque siempre se debe tratar de lograr mucha cohesión, con frecuencia es necesario y aconsejable hacer que un componente de software realice funciones múltiples. Sin embargo, para lograr un buen diseño hay que evitar módulos que lleven a cabo funciones no relacionadas.

El acoplamiento es una indicación de la interconexión entre módulos en una estructura de software, y depende de la complejidad de la interfaz entre módulos, del grado en el que se entra o se hace referencia a un módulo y de qué datos pasan a través de la interfaz. En el diseño de software, debe buscarse el mínimo acoplamiento posible. La conectividad simple entre módulos da como resultado un software que es más fácil de entender y menos propenso al “efecto de oleaje”, ocasionado cuando ocurren errores en un sitio y se propagan por todo el sistema.

## Herramientas de diseño

Las *herramientas de diseño* son instrumentos que ayudan a representar los modelos de diseño del software. Algunas de las más usuales son:

*Diagramas UML*: De clase, de interacción, de paquetes, de despliegue, etc.

*Cartas de estructura (diagramas de estructura).* Permiten representar gráficamente la arquitectura modular de un sistema estructurado. Además, muestran también información sobre la secuencia de ejecución, control y datos enviados o recibidos

*Tabla de decisión (tablas de transición).* Definen, en forma matricial, reglas con acciones a realizar dadas ciertas condiciones.

*Diagramas de flujo de control (organigramas estructurados*). Permiten la representación gráfica de un proceso. Cada paso del proceso se expresa mediante un símbolo diferente que contiene una breve descripción de la etapa del proceso. Los símbolos gráficos están unidos entre sí con flechas que indican la dirección del flujo del proceso.

*Diagramas de Nassi-Shneiderman (diagramas NSD o diagramas de Chaplin).* Representación gráfica que muestra el diseño de un programa estructurado. Basado en un diseño descendente, refleja la descomposición de un problema de forma simple usando cajas anidadas para describir cada uno de los subproblemas. Son isomórficos con los diagramas de flujo de control.

*Lenguajes de diseño de programas (LDP, lenguaje estructurado o pseudocódigo).* Lenguaje que utiliza el vocabulario de un lenguaje y la sintaxis de otro. Independientemente de su origen, un LDP tiene: (1) *una sintaxis fija de palabras*

*clave*, para definir construcciones estructuradas, declarar datos y establecer características de modularidad, (2) *una sintaxis libre en lenguaje natural*, para describir características de procesamiento, (3) *facilidades para declarar datos*, incluyendo estructuras de datos simples y complejas, y (4) *un mecanismo de definición de subprogramas* y de llamadas a éstos.

## Métodos de diseño

Un *método de diseño* proporciona las herramientas, las técnicas y los pasos a seguir para obtener diseños de forma sistemática. Independientemente del método de diseño que se use, todos tienen varias características comunes (ver transparencia 17 del tema 3.1)

Los principales métodos de diseño son:

*Diseño Estructurado de Sistemas (SSD*). Orientado a la identificación, selección y organización de los módulos y sus relaciones. A partir del resultado del proceso de análisis, se lleva a cabo una descomposición del sistema en módulos estructurados en jerarquías y con características que permiten la implementación de un sistema que no requiere elevados costes de mantenimiento.

*Desarrollo de Sistemas Jackson (JSD)*. Se aplican los siguientes pasos. (1) Identificación de *entidades* (elementos que un sistema necesita para producir o utilizar información) y *acciones* (sucesos que afectan a las entidades). (2) Identificación de la estructura de las entidades: las acciones que afectan a cada entidad se ordenan por tiempo y se representan mediante diagramas. (3) Modelado inicial: las estructuras y acciones se representan como un modelo de proceso y se definen las conexiones entre este modelo y el mundo real. (4) Planificación del sistema: se especifican las características de planificación del proceso; y (5) Implementación: el hardware y el software se integran como un proceso.

*Entidad-Relación-Atributo (ERA)*. Es un modelo de datos basado en una percepción del mundo real que consiste en un conjunto de objetos básicos llamados entidades (descritos mediante un conjunto de atributos) y relaciones entre estos objetos, implementándose en forma gráfica a través del Diagrama Entidad-Relación.

*Técnicas de Modelado de Objetos (OMT).* Se basan en un conjunto de conceptos que definen qué es la orientación a objetos, y en una notación gráfica independiente que ayuda al desarrollo de software visualizando el problema sin recurrir, de forma prematura, a la implementación. OMT utiliza tres modelos. (1) *Modelo de objetos*: representación, mediante diagramas de clase y diagramas de casos concretos, de objetos, clases y relaciones. (2) *Modelo dinámico*: representación, mediante diagramas de estados, de los aspectos relacionados con el tiempo y con los cambios, (3) *Modelo funcional*: descripción, mediante diagramas de flujo de datos, los cálculos existentes dentro del sistema.

*Método de Booch*. Plantea trabajar en dos partes básicas: un *micro-proceso* y un *macro-proceso*. Ambas partes incluyen varios pasos: identificación de clases y objetos a un nivel de abstracción dado, identificación de la semántica de esas clases y objetos, identificación de las relaciones entre clases y objetos, selección de la estructura de datos y algoritmos para la implementación de esas clases y objetos, la conceptualización del sistema, etc. Este método proporciona una forma de desarrollar análisis y diseño de un sistema orientado a objetos. Se enfoca en el análisis y el diseño y no en la implementación o las pruebad. Define seis tipos de diagramas: clase, objeto, estado de transición, interacción, módulo y proceso.

*Métodos orientados a objetos*. Actualmente existe una gran variedad de métodos de este tipo aunque la mayoría de ellos utilizan como herramienta de modelado UML y como proceso de desarrollo el UP (Proceso Unificado)

# Modelo de diseño

En el análisis, el objetivo era crear un modelo lógico que capturara la funcionalidad que el sistema debía proporcionar para satisfacer los requisitos del usuario. En el diseño, la finalidad es especificar completamente cómo se implementará esta funcionalidad. Una forma de examinar esto es considerar el ámbito del problema por un lado y el ámbito de la solución por otro. Los requisitos provienen del ámbito del problema, y se puede considerar el análisis como una exploración de ese ámbito desde el punto de vista de los usuarios del sistema. El diseño implica perderse en soluciones técnicas del ámbito de la solución para proporcionar un modelo del sistema (el modelo de diseño) que se pueda implementar.

## Contenido del modelo

El modelo de diseño contiene muchos subsistemas de diseño (ver transparencia 19 del tema 3.1). Estos subsistemas son componentes que pueden contener muchos tipos diferentes de elementos de modelado.

Se puede pensar en el modelo de diseño como una elaboración del modelo de análisis con detalles añadidos y soluciones técnicas específicas. El modelo del diseño contiene los mismos elementos que el modelo del análisis, pero todos los elementos están mejor formados y deben incluir detalles de implementación. Por ejemplo, una clase de análisis puede ser poco más que un boceto con algunos atributos. Sin embargo, una clase de diseño debe estar especificada completamente, todos los atributos y operaciones (incluidos tipos de retorno y listas de parámetros) deben estar completos.

Los modelos de diseño constan de: Subsistemas de diseño, clases de diseño, interfaces, realizaciones de caso de uso-diseño, y un diagrama de despliegue.

Uno de los elementos básicos que genera el diseño son las interfaces, las cuales permiten desacoplar el sistema en subsistemas que se pueden desarrollar en paralelo. El diseño también genera un diagrama de despliegue que muestra cómo se distribuye el sistema software sobre nodos computacionales físicos. Claramente, éste es un diagrama importante y estratégico.

## Relación con el modelo de análisis

Existe una sencilla relación `<<trace>>` entre los modelos de análisis y diseño: el modelo de diseño está basado en el modelo de análisis y se puede considerar como una mejora de él (ver transparencia 21 del tema 3.1)

La relación entre los paquetes de análisis y los subsistemas de diseño puede ser compleja. Algunas veces un paquete de análisis hará un `<<trace>>` a un subsistema de diseño, pero éste no siempre es el caso. Puede haber buenas razones técnicas y de arquitectura para desglosar un solo paquete de análisis en más de un subsistema.

Una clase de análisis se puede resolver en una o más interfaces o clases de diseño. Esto es porque las clases de análisis son una vista conceptual de alto nivel de las clases del sistema. Cuando se pasa al modelado físico (diseño), estas clases conceptuales se pueden implementar como una o varias clases físicas de diseño y/o interfaces.

La realización de caso de uso – análisis tiene una realización <<trace>> uno a uno con realización de caso de uso – diseño. En diseño la realización de caso de uso simplemente tiene más detalle.

¿Se deberían mantener dos modelos? En un mundo ideal, se debería tener un solo modelo del sistema y una herramienta de modelado que fuese capaz de proporcionar una vista de análisis de ese modelo o una vista de diseño. Sin embargo, éste es un requisito más difícil de lo que parece a primera vista y ninguna herramienta de modelado UML actualmente en el mercado realiza un trabajo enteramente satisfactorio para proporcionar vistas de análisis y diseño del mismo modelo subyacente. En la transparencia 20 del tema 3.1, se muestran cuatro posibles estrategias.

No existe mejor estrategia, depende del proyecto. Sin embargo, la pregunta fundamental que se necesita formular es "¿Es necesario conservar una vista de análisis del sistema?" Las vistas de análisis proporcionan la visión de conjunto del sistema. Una vista de análisis puede tener solamente entre un 1 por ciento y un 10 por ciento de las clases que están en la vista detallada de diseño y son mucho más entendibles.

En general, se debe conservar una vista de análisis para cualquier sistema que sea amplio, complejo, estratégico o potencialmente de larga duración. Esto significa elegir entre la estrategia 3 y 4. Siempre hay que pensar si permitir que los modelos de análisis y diseño no vayan al mismo ritmo. ¿Es esto aceptable para el proyecto?

Si el sistema es pequeño (menos de 200 clases de diseño), entonces el propio modelo de diseño es suficientemente pequeño para ser entendible por lo que un modelo de análisis aparte puede no ser necesario. Igualmente, si el sistema no es estratégico o tiene un ciclo de vida proyectado corto, modelos aparte de análisis y diseño puede ser peligroso.

La elección está entonces entre la estrategia 1 y 2 y el factor decisivo será la posibilidad de uso de la herramienta de modelado de UML. Algunas herramientas de modelado mantienen un solo modelo subyacente y permiten filtrar y ocultar información para tratar de recuperar una vista de "análisis" del modelo de diseño. Ésta es una opción razonable para muchos sistemas de tamaño medio, pero probablemente no es suficiente para sistemas muy grandes. Por último, una llamada a la precaución; es acertado recordar que muchos sistemas sobreviven más tiempo que el ciclo de vida proyectado.