

# Desarrollo de software

## Objetivos

- Comprender el concepto de software.
- Diferenciar los conceptos de código fuente, código objeto y código ejecutable.
- Clasificar los lenguajes de programación.
- Conocer el proceso de obtención del código ejecutable a partir del código fuente y las herramientas implicadas.
- Conocer el concepto de máquina virtual y su utilidad.
- Comprender la razón de ser de la ingeniería del software.
- Identificar las fases del desarrollo de una aplicación informática y el cometido de cada una de ellas.
- Conocer los diferentes modelos de ciclo de vida del software.
- Conocer las metodologías de desarrollo de software actuales.

## Contenidos

- 1.1. El software y su relación con otras partes del ordenador
- 1.2. Lenguajes de programación. Tipos
- 1.3. Código fuente, código objeto y código ejecutable. Herramientas implicadas
- 1.4. Máquinas virtuales
- 1.5. La ingeniería del software
- 1.6. Fases del desarrollo de una aplicación informática
- 1.7. Roles que intervienen en el proceso de desarrollo de software
- 1.8. Modelos de ciclo de vida del software
- 1.9. Metodologías de desarrollo de software

## Introducción

Sería imposible concebir la sociedad actual sin tener en cuenta la presencia de los ordenadores, pues estos desempeñan un papel trascendental en muchos ámbitos de nuestra actividad cotidiana. En esta unidad, **se analiza el software del ordenador** y el **proceso que es necesario llevar a cabo para la creación de programas o aplicaciones informáticas**. Este es el **objeto de estudio de una disciplina conocida como Ingeniería del software**.

Una de las etapas fundamentales del desarrollo de software es la de programación, tarea en la que entran en juego los lenguajes de programación. Por ello, aquí se lleva a cabo una **clasificación de estos lenguajes** y **se analiza el proceso que parte del código fuente hasta el código ejecutable y las herramientas necesarias para realizar esta tarea**.

### 1.1. El software y su relación con otras partes del ordenador

La definición de **software** que proporciona la Real Academia Española (RAE) es la siguiente: «conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora».

Según la definición de Pressman (2010), más rigurosa y técnica, **el software es:** «1) Instrucciones (programas de cómputo) que cuando se ejecutan proporcionan las características, función y desempeño buscados; 2) estructuras de datos que permiten que los programas manipulen de forma adecuada la información, y 3) información descriptiva tanto en papel como en formas virtuales que describen la operación y uso de los programas.»

En cualquier caso, se hace referencia a los programas que se ejecutan en un ordenador con el fin de realizar determinadas tareas sobre el hardware, y los datos necesarios para la ejecución de dichos programas.

Para comprender un poco mejor el concepto de software, es necesario hacer referencia a **las diferentes partes que componen el hardware de un ordenador**, que son básicamente tres:

1. **Unidad central de proceso (CPU):** es la parte del ordenador que ejecuta las instrucciones contenidas en los programas. Aunque las instrucciones de un programa puedan ser muy complejas, estas al final **se traducen o materializan en sencillas operaciones aritméticas (sumas, restas, etc.) y lógicas (OR, AND, etc.) que se realizan sobre bits**. La CPU consta, a su vez, de las siguientes partes:
  - **Unidad aritmético-lógica (ALU):** ejecuta las operaciones aritméticas y lógicas encendidas por la unidad de control con los datos que recibe, y devuelve el resultado de dichas operaciones, siguiendo las órdenes de la unidad de control.
  - **Unidad de control (UC):** recoge las instrucciones contenidas en la memoria principal y ordena su ejecución mediante el envío de señales a la ALU y a los registros.

- **Registros:** constituyen el almacenamiento interno de la CPU e intervienen en la ejecución de las instrucciones. Hay varios registros, como el contador de programa y el registro de instrucción.
2. **Memoria principal o memoria RAM:** contiene las instrucciones del programa que hay que ejecutar y los datos sobre los que deben operar estas instrucciones. La CPU toma estas instrucciones de la memoria RAM y envía las órdenes necesarias para su ejecución. Se trata de una memoria volátil, lo que quiere decir que su contenido desaparece cuando se apaga el ordenador.
  3. **Unidad de entrada/salida:** permite la comunicación del ordenador con el exterior transfiriendo la información a través de periféricos. Los periféricos pueden ser de varios tipos:
    - **De entrada:** proporcionan al ordenador datos e instrucciones. Ejemplos: teclado, ratón, etcétera.
    - **De salida:** muestran información al exterior, es decir, al usuario del ordenador. Ejemplos: pantalla, impresora, etcétera.
    - **De entrada/salida:** proporcionan información al ordenador y envían información del ordenador al exterior. Ejemplos: módem, tarjeta de red, etc. También se puede incluir entre estos periféricos los dispositivos que permiten almacenar información de manera permanente (memoria secundaria), como los discos duros, las memorias flash, DVD, etcétera.

Para la interconexión de todos estos elementos, existe una serie de conexiones que reciben el nombre de **buses**, que se representan mediante flechas (véase Figura 1.1).

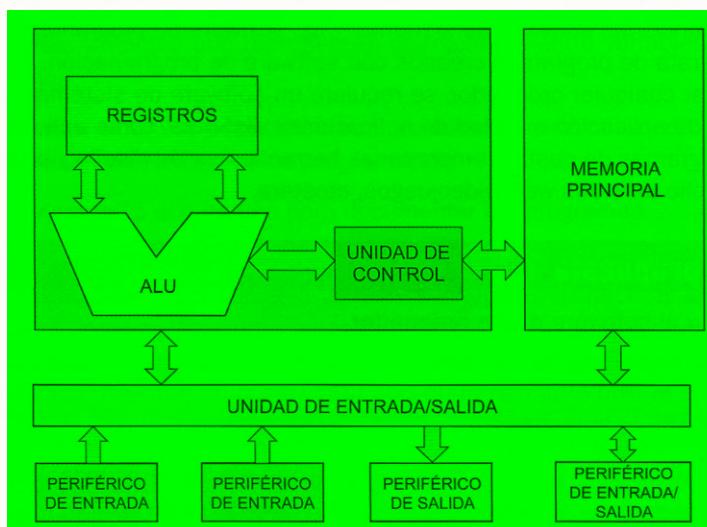


Figura 1.1. La estructura de un ordenador consta de una CPU, dividida en: unidad de control, unidad aritmético-lógica (ALU) y registros; la memoria principal o memoria RAM y la unidad de entrada/salida, que permite la comunicación del ordenador con el exterior gracias a la conexión de los periféricos con esta unidad. Los diversos elementos del hardware de un ordenador se conectan por medio de unas líneas llamadas buses por las que circulan los bits.

El software, por tanto, hace referencia al conjunto de instrucciones que debe ejecutar el ordenador para resolver un problema a partir una serie de datos de entrada con el fin de obtener determinados resultados. Tanto estas instrucciones como los datos sobre los que estas operan deben residir en la memoria principal o memoria RAM. Estas instrucciones son tomadas por la unidad de control, la cual envía las órdenes necesarias con el fin de que la ALU realice las operaciones para dar respuestas a esas instrucciones. Los datos de entrada para resolver el problema pueden ser suministrados al ordenador por medio de una serie de periféricos de entrada, como el teclado y el ratón, y los resultados se pueden mostrar mediante los periféricos de salida, como la pantalla y la impresora. La información que se requiere guardar de manera permanente se almacena en periféricos de entrada/salida, como el disco duro o una memoria flash.

Existen distintos tipos de software en función de su uso:

- **Software de sistemas:** se trata de programas que permiten el uso de un ordenador para lo cual se comunican con el hardware y permiten la interacción entre el usuario y el ordenador. Son ejemplos de software de sistemas, los sistemas operativos, los controladores o drivers de dispositivos y las herramientas de diagnóstico.
- **Software de programación o desarrollo:** posibilita la creación de programas. Los editores, compiladores o depuradores, entre otros, son ejemplos de este tipo de software. En el Apartado 1.3, se tratarán las herramientas para la creación de programas. Sin embargo, lo más habitual es usar un tipo de software que integre todas estas herramientas, los llamados entornos de desarrollo integrados.
- **Software de aplicación:** se trata de todo aquel software que no se incluye en los grupos anteriores. Son todos aquellos programas que ayudan a las y los usuarios a realizar algún tipo de tarea y que permiten que el ordenador sea un objeto útil para las personas que lo usan. La existencia de este tipo de software es posible gracias a la existencia del software de sistemas y del software de programación, ya que, al final, se trata de programas creados con software de programación, y porque, para poder usar cualquier ordenador, se requiere un software de sistemas. Constituyen software de aplicación multitud de aplicaciones distintas, como aplicaciones ofimáticas, programas de gestión empresarial, herramientas de diseño y programas para el desarrollo de sitios web, videojuegos, etcétera.

## Actividad propuesta 1.1

### El hardware y el software de un ordenador

Relaciona los elementos de hardware o software de un ordenador que aparecen en la columna de la izquierda con el tipo de hardware o de software correspondiente de la columna de la derecha:

Procesador de textos
Impresora
Ratón
Compilador
Sistema operativo
Memoria flash

Periférico de entrada
Periférico de salida
Periférico de entrada/salida
Software de sistemas
Software de programación
Software de aplicación

## 1.2. Lenguajes de programación. Tipos

Las instrucciones de los programas que ha de ejecutar el ordenador para resolver un determinado problema deben estar escritas en el lenguaje que comprenda el ordenador, que es el lenguaje binario (compuesto por ceros y unos), pero sería enormemente complicado para un programador escribir sus programas empleando únicamente ceros y unos. Como se verá más adelante, quienes programan pueden utilizar **lenguajes mucho más cercanos al lenguaje natural que facilitan enormemente la creación de programas para resolver problemas complejos**. Para ello, emplearán los **llamados lenguajes de programación de alto nivel**. De hecho, es posible realizar una clasificación de los lenguajes de programación en función de su cercanía a la máquina, esto es, en función de la medida en que ese lenguaje es próximo al lenguaje binario directamente comprensible por un ordenador. No obstante, antes de proceder a describir dicha clasificación, en este apartado, se caracterizarán los lenguajes de programación.

**Un lenguaje de programación puede definirse como una notación para escribir programas a través de los cuales es posible establecer una comunicación con el hardware y dar así las órdenes necesarias para la realización de una determinada tarea.** Los lenguajes de programación vienen definidos por una gramática o conjunto de reglas que se aplican a un alfabeto compuesto por un conjunto de símbolos.

Los **elementos básicos de los lenguajes de programación** son los siguientes:

- **Identificadores:** nombres simbólicos que se dan a ciertos elementos de programación (variables, tipos, módulos, etc.).
- **Constantes:** datos que no cambian su valor a lo largo del programa.
- **Operadores:** símbolos que representan operaciones entre variables, constantes y expresiones.
- **Instrucciones:** símbolos especiales que representan estructuras de procesamiento y de definición de elementos de programación.
- **Comentarios:** texto que se usa para documentar los programas.

Como se indicó anteriormente, se puede realizar una **clasificación de los lenguajes de programación atendiendo a la cercanía de dichos lenguajes respecto al lenguaje que utiliza el ordenador (lenguaje binario) o respecto al lenguaje de las personas o lenguaje natural**. Sobre la base de este parámetro, es posible establecer tres grupos de lenguajes:

- **Lenguajes de bajo nivel o lenguajes máquina:** el lenguaje máquina es el único lenguaje de programación que entiende directamente la máquina. Utiliza el alfabeto binario (números 0 y 1) para establecer la comunicación con el hardware de la máquina. Fue el primer lenguaje empleado para la programación de ordenadores, pero dejó de usarse debido a su complejidad, siendo sustituido por otros más cercanos al lenguaje natural, que eran más fáciles de aprender y de utilizar. **Las instrucciones en el lenguaje máquina son secuencias de unos y ceros.** Para referirse a los datos que se van a procesar, este lenguaje utiliza la dirección de memoria en la que se encuentran, de forma que es tarea del programador asignar una dirección a cada dato.

■ **Lenguajes intermedios o lenguajes ensambladores:** con estos lenguajes ya no se utilizan códigos numéricos ni direcciones reales de memoria, sino que se sustituyen por **representaciones textuales, equivalentes a las instrucciones del lenguaje máquina que representan**, es decir, **se codifican las instrucciones de forma simbólica**. A cada instrucción o dato, le corresponde un código mnemotécnico, cuya formulación guarda alguna relación con la operación o el dato al que representa (por ejemplo, pueden ser abreviaturas). Estos lenguajes utilizan direcciones simbólicas en lugar de direcciones binarias para referirse a los datos. Los lenguajes ensambladores guardan una estrecha relación con los lenguajes máquina, ya que **cada instrucción de los programas en lenguaje ensamblador corresponde a una instrucción en lenguaje máquina**; la diferencia radica en la forma de codificación o representación de los lenguajes ensambladores, que es mucho más fácil de entender por el hombre. **Con el fin de que el ordenador entienda las instrucciones escritas en lenguaje ensamblador, estas se traducen a lenguaje máquina (secuencias de ceros y unos).** Para ello **se emplea un traductor, que también recibe el nombre de ensamblador**. En la Figura 1.2 se muestra una porción de código en lenguaje ensamblador y su correspondencia en lenguaje máquina. De este proceso de transformación se encarga un ensamblador.



Figura 1.2. Las instrucciones en lenguaje ensamblador están escritas empleando códigos mnemotécnicos (add indica sumar, mov, mover, etc.) y cada instrucción corresponde a una instrucción en lenguaje máquina. Para pasar del lenguaje ensamblador al lenguaje máquina se usa un traductor llamado ensamblador.

■ **Lenguajes de alto nivel o lenguajes evolucionados:** tienen como objetivo principal liberar al programador de tareas tediosas y complejas que pueden frenar su productividad y eficiencia. Esto se consigue gracias al **gran nivel de abstracción que ofrecen estos lenguajes**, lo cual **hace que sea innecesario que las programadoras y programadores conozcan las características de la máquina**. Otra ventaja es la proximidad que presentan estos lenguajes respecto al lenguaje natural. Sus **características** más importantes son las siguientes:

- **Las instrucciones se expresan por medio de caracteres alfanuméricos, numéricos y especiales.**
- **Se pueden definir variables para recoger los datos que se vayan a tratar.** Estas variables pueden adoptar la forma de complejas estructuras de datos, lo que facilita enormemente la labor de programación.
- **Pueden incluirse líneas de comentarios.**
- **Disponen de instrucciones muy potentes de tipo aritmético, lógicas, de tratamiento de caracteres, etcétera.**

- El tiempo de codificación y puesta a punto de los programas es mucho menor que el requerido por los anteriores lenguajes.
- Resultan más fáciles de corregir y modificar, lo que conlleva un gran ahorro de tiempo de programación.
- La curva de aprendizaje de las personas responsables de la programación es más corta y menos pronunciada que con los lenguajes máquina y ensambladores.

Sin embargo, no todo son ventajas: estos programas también presentan inconvenientes:

- Los programas escritos en estos lenguajes no pueden ejecutarse directamente, por lo que existe un tiempo durante el cual se deben traducir, tiempo que los anteriores no requieren.
- La ocupación de memoria se incrementa, tanto por el traductor como por el programa traducido. No obstante, hoy en día, las posibilidades que nos ofrece el hardware y los sistemas operativos restan importancia a esta cuestión.
- La eficiencia en el consumo de recursos y las prestaciones conseguidas son bastante menores que con los lenguajes máquina y ensambladores.



Figura 1.3. Nombres de algunos lenguajes de programación de alto nivel.

En el ámbito de los lenguajes de alto nivel, también es posible realizar una clasificación en función del paradigma de programación empleado. Así, cabe distinguir entre lenguajes estructurados y lenguajes orientados a objetos.

Los lenguajes de programación estructurados se caracterizan por el empleo exclusivo de tres estructuras para la creación de cualquier programa: la estructura secuencial, la alternativa y la repetitiva. Además, en la metodología estructurada, para la que se emplean los lenguajes estructurados, las aplicaciones informáticas integran diversos componentes de software llamados módulos. Un módulo realiza determinadas tareas de procesamiento, para lo que requiere utilizar determinados datos. Por lo tanto, una aplicación informática se compone de diversos módulos que realizan pequeñas tareas. Los

módulos se combinan entre sí, realizándose llamadas de unos a otros. En estas llamadas, los módulos se pasan datos (llamados parámetros) y a veces devuelven resultados. Los módulos pueden ser de dos tipos: procedimientos o funciones. La diferencia entre estos es que las funciones devuelven algún dato como resultado de su ejecución, mientras que los procedimientos no lo hacen.

## Argot técnico



En algunos lenguajes de programación, como C, no se realiza de manera explícita la distinción entre procedimiento y función, sino que se llama función a cualquier módulo. En estos lenguajes, una función puede devolver un resultado o no devolver nada, de forma que se puede considerar que, conceptualmente, las funciones que no devuelven ningún resultado son en realidad procedimientos.

La arquitectura de una aplicación informática consta de un módulo de control principal que llama a varios módulos subordinados. Estos módulos llaman a otros y así sucesivamente. La arquitectura de un programa se puede representar mediante lo que es conocido como diagrama de estructuras, en el que cada rectángulo representa un módulo y las flechas que unen los módulos representan las llamadas. Estas llamadas parten de un módulo y llegan a otro. En el diagrama de estructuras de la Figura 1.4, el módulo de control principal se llama A y este módulo llama a los módulos subordinados B y C; a su vez, B llama al D y al E, y el C llama al F.

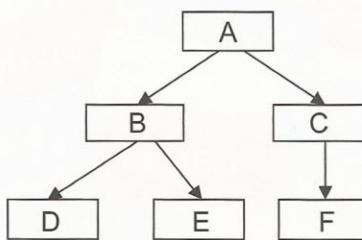


Figura 1.4. Diagrama de estructuras que muestra la arquitectura de una aplicación informática siguiendo la metodología estructurada.

En la metodología estructurada, la atención se centra en los procesos o tareas que deben realizarse para resolver un determinado problema y estos procesos o tareas se distribuyen en módulos que los ejecutan. Para llevar a cabo estos procesos, es necesario que los módulos manipulen ciertos datos. Los datos que se tienen que almacenar de manera permanente se registran en una base de datos, sobre la que los diferentes módulos realizan operaciones de consulta, inserción, modificación o borrado.

Son ejemplos de lenguajes de programación estructurados: Pascal, C, COBOL.

Los lenguajes orientados a objetos, que surgieron más tarde, supusieron en cierto modo un cambio de visión en relación con el desarrollo de software. Se pasó de un enfoque centrado en los procesos a prestar mayor atención a los datos sobre los que había que operar. Según el paradigma orientado a objetos, se considera que una aplicación

informática consta de diversos objetos que interactúan entre sí. Un objeto es una instancia de una clase, la cual está constituida por una serie de datos y un conjunto de operaciones que se aplican sobre dichos datos. Los datos de una clase reciben el nombre de atributos y las operaciones que se aplican sobre esos datos se llaman métodos. La ejecución de todas las tareas encomendadas a un programa se logra mediante la participación cooperativa de los objetos que componen la aplicación. Si en las metodologías estructuradas, esta participación cooperativa se conseguía mediante las llamadas de unos módulos a otros, en el paradigma orientado a objetos, esta se consigue mediante el envío de mensajes. Un mensaje representa la solicitud enviada desde un método de un objeto para que otro objeto o él mismo ejecute otro método.

Por otra parte, un objeto se identifica por un conjunto de datos que configuran su estado y un conjunto de métodos que indican el comportamiento que muestra dicho objeto:

- El estado de un objeto viene determinado por el conjunto de propiedades o atributos que tiene el objeto y los valores que toman estos en cada momento. En el sistema de gestión de una biblioteca, un objeto puede ser un libro en concreto y su estado puede devenir definido por los atributos: código, ISBN, título, editorial, número de páginas y su situación de prestado o no; junto con los valores que toma cada uno de estos atributos para ese libro en concreto, por ejemplo: A087 para el código; el número 1234567890123 sería el valor del ISBN; el título tomaría como valor «Introducción a la programación»; el valor de la editorial sería «Paraninfo», con 140 páginas y el valor de la situación de prestado sería «falso». En el sistema de gestión de una entidad bancaria, un objeto puede ser una cuenta bancaria en concreto y su estado puede estar definido por los atributos número de cuenta y saldo. Los valores que tomarían estos atributos para una cuenta en concreto serían, por ejemplo, ES1200781234547708669999 para el número de cuenta y 4.323,65 euros para el saldo.
- El comportamiento de un objeto está determinado por la manera en que actúa a recibir un mensaje por parte de otro objeto, posiblemente cambiando su estado. La manera de responder a un mensaje es mediante la ejecución de una operación que está asociada a dicho objeto.

Algunos ejemplos de lenguajes orientados a objetos son Java, C# y C++.

## Actividad propuesta 1.2

### Lenguajes de programación estructurados y orientados a objetos

Como sabes, se puede realizar una clasificación de los lenguajes de programación de alto nivel en función de si permiten o no realizar programación orientada a objetos. Así, los que no la permiten se llaman *lenguajes estructurados*, y los que sí, *lenguajes orientados a objetos*.

Investiga si los siguientes lenguajes de programación son estructurados u orientados a objetos: Python, Ruby, PHP, Pascal, Swift, Smalltalk, Scala.

Para el caso de los lenguajes estructurados, investiga cuáles de ellos han incorporado la programación orientada a objetos dando lugar a un lenguaje orientado a objetos, tal y como ocurrió con el lenguaje orientado a objetos C++, que surgió como extensión del lenguaje estructurado C.

## ■ 1.3. Código fuente, código objeto y código ejecutable. Herramientas implicadas

Las instrucciones de los programas que ejecutan los ordenadores para resolver un determinado problema deben estar escritas en el lenguaje que estos comprenden, que es el lenguaje binario. Pero, como se ha señalado antes, programar en este lenguaje binario es una labor compleja y muy tediosa, por lo que, actualmente, se utilizan para ello lenguajes de programación de alto nivel. Por este motivo, es necesario un proceso que transforme el programa escrito en un lenguaje de alto nivel en otro escrito en lenguaje máquina o lenguaje binario. Es en este proceso en el que se puede hablar de códigos fuente, objeto y ejecutable.

Se considera **código fuente** aquel que el programador escribe directamente, por lo que en casi todos los casos, **consiste en instrucciones escritas siguiendo las normas de un lenguaje de programación de alto nivel**. **Como el ordenador no comprende este lenguaje, debe ser transformado en código objeto**. Dependiendo de cómo se lleve a cabo este proceso de traducción o de transformación del código fuente en código objeto, hay dos tipos de traductores:

1. **Compiladores**. **Son traductores que realizan su tarea de forma global**, esto es, **en un único proceso se analiza todo el programa fuente, se genera el código objeto respectivo y se almacena el resultado**. Todo esto se podrá llevar a cabo en caso de que el programa fuente esté correctamente escrito de acuerdo con las normas del lenguaje de programación correspondiente; en caso de que no sea así, no se generará el código objeto y se mostrarán uno o varios mensajes de error que pueden ayudar a la persona encargada de la programación a corregir estos errores. **El código objeto generado** como resultado de la compilación, dependiendo del tipo de compilador, **se podrá ejecutar directamente o puede que se precisen otros pasos antes de que el programa sea ejecutable, como los pasos de ensamblado, enlazado y carga**. Una vez obtenido **el código ejecutable**, este **se podrá ejecutar tantas veces como se desee sin necesidad de tener que volver a realizar el proceso de compilación**.
2. **Intérpretes**. A diferencia de lo que ocurre con los compiladores, estos simultánean el proceso de traducción y el de ejecución. La forma de trabajo de los intérpretes es la siguiente: **analizan bloques del programa fuente, generan el código objeto correspondiente y lo ejecutan, luego repiten este proceso hasta que acaba el programa**. En la etapa de ejecución simplemente se pasa el control al código objeto generado y se espera a que termine su ejecución. De esta forma, **cada fragmento de código fuente solo se almacena provisionalmente**.

En el caso del lenguaje Java, se combinan la compilación y la interpretación: el programa se compila en primer lugar, luego se genera un formato intermedio llamado **bytecode** y este resultado es interpretado por la máquina virtual de Java.

Para realizar todo el proceso, desde la generación del código fuente hasta la ejecución del programa, se precisa una serie de herramientas:

- Para escribir el código fuente se puede emplear un editor de textos simple o alguna herramienta de programación. Hay programas, como *Notepad++* o *Sublime*, que permiten crear código fuente en diferentes lenguajes de programación y facilitan el trabajo de las y los programadores, marcando los diferentes elementos del código fuente en distintos colores, lo que facilita de manera importante la tarea de escritura del código fuente.
- En función del lenguaje de programación utilizado, para la transformación del código fuente en código objeto, se empleará un compilador o un intérprete.
- También puede ocurrir que el código objeto generado no sea ejecutable directamente, en cuyo caso, será necesario emplear alguna herramienta adicional, como un enlazador. El enlazador inserta en el código objeto una serie de rutinas y bibliotecas que permiten que el código sea directamente ejecutable por el ordenador.

No obstante, en lugar de utilizar estas herramientas de manera independiente, en la actualidad, se suele emplear un entorno de desarrollo integrado, que es una aplicación informática compuesta por un conjunto de herramientas que facilitan a la persona encargada de la programación su tarea y posibilitan una mayor rapidez en la creación de programas. La mayoría de los entornos de desarrollo incluyen, además de un editor de textos con ayudas visuales y un compilador o un intérprete, otras herramientas, como un depurador (para ayudar en la detección y corrección de errores de programación), una herramienta de control de versiones y un constructor de interfaz gráfica.

## 1.4. Máquinas virtuales

Una máquina virtual es una aplicación que ejecuta los programas como si fuese una máquina real, aunque no lo sea. Existen dos tipos de máquinas virtuales:

1. **Máquinas virtuales de sistema:** son aplicaciones que emulan a un ordenador por completo, de forma que se puede instalar en su interior otro sistema operativo con su propio disco duro, memoria, tarjeta gráfica, etc. Se puede trabajar en la máquina virtual de igual forma que si se tratase de una máquina real. Son ejemplos de aplicaciones de este tipo *VMware Workstation* y *Virtual Box*. Se pueden utilizar estas máquinas virtuales para disponer de manera sencilla de varios sistemas operativos en el mismo ordenador (instalando uno de ellos en la máquina real y otro diferente en la virtual). Esto permite, por ejemplo, evaluar nuevos sistemas operativos y probar aplicaciones en diferentes sistemas operativos. Sin embargo, es importante tener en cuenta que las máquinas virtuales suponen una carga importante para el ordenador físico en el que se instalan, por lo que se requiere que este tenga una capacidad notable, sobre todo, en cuanto a disco duro y memoria RAM.

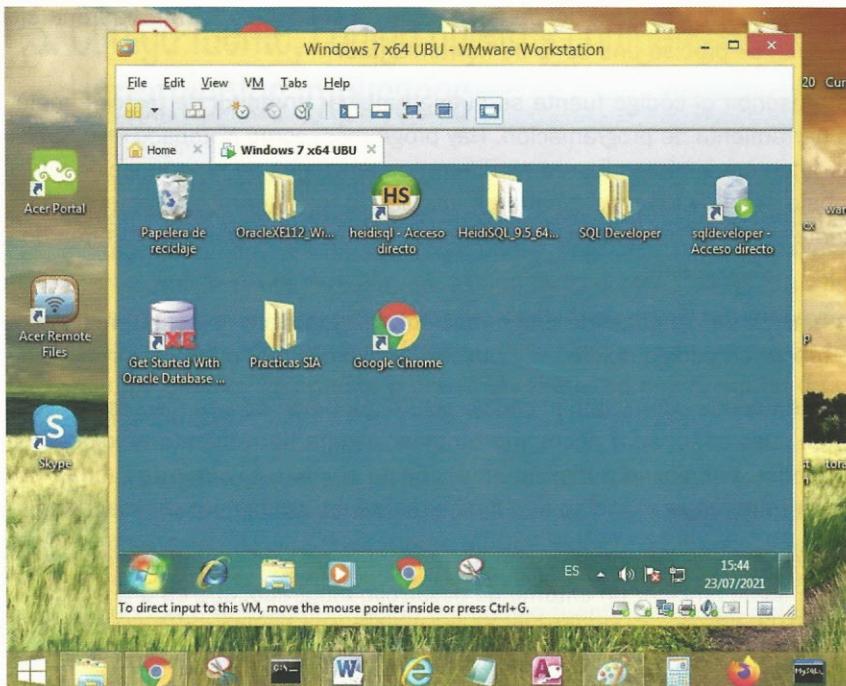


Figura 1.5. Uso del programa VMware Workstation para poder utilizar, dentro de un equipo, otro equipo que tiene incorporado su propio sistema operativo y disco duro, entre otras tecnologías.

2. **Máquinas virtuales de proceso:** estas máquinas ejecutan un proceso concreto dentro de un sistema operativo. Este tipo de máquinas se inician cuando se lanza el proceso que se desea ejecutar y se detienen cuando este termina. El objetivo de estas máquinas es permitir que un programa se ejecute de igual forma en cualquier plataforma, proporcionando un entorno de ejecución independiente del hardware y del sistema operativo y ocultando los detalles de la plataforma subyacente. El ejemplo más popular de máquina virtual de proceso es la máquina virtual de Java.

La máquina virtual de Java (JVM, por sus siglas en inglés, Java virtual machine) es el ejemplo típico de máquina virtual de proceso. Su ventaja es que dota de portabilidad al lenguaje, de manera que un programa compilado en Java se puede ejecutar en cualquier plataforma. Esto se debe a que el programa escrito en Java realmente no es ejecutado por el procesador del ordenador, sino por la JVM.

## Recuerda

La máquina virtual de Java (JVM), como se estudiará en la Unidad 2, forma parte del entorno en tiempo de ejecución de Java (JRE, por sus siglas en inglés, Java Runtime Environment), que está constituido por un conjunto de utilidades que permite la ejecución de programas en Java. El JRE incluye, además de la JVM, un conjunto de librerías de Java y otros componentes necesarios para que un programa escrito en Java se pueda ejecutar.



El proceso que se lleva a cabo para poder ejecutar un programa escrito en Java es el siguiente:

1. La persona encargada de la programación escribe el código fuente del programa en lenguaje Java mediante el empleo de un editor de textos, para lo que puede usar el que viene incorporado en el entorno de desarrollo integrado. De esta manera, se genera un archivo de texto con extensión .java.
2. Se compila el programa fuente mediante el compilador `javac`, que también viene incorporado en el entorno de desarrollo, lo que da lugar a un fichero con extensión `.class`, en caso de que no haya errores en el código fuente. Este fichero contiene código en un lenguaje intermedio llamado `bytecode`, que es independiente del ordenador y del sistema operativo en el que se ejecuta.
3. La máquina virtual de Java traduce el archivo con extensión `.class` al código binario para que el programa pueda ser ejecutado. Como la JVM está disponible en diferentes sistemas operativos, los archivos con extensión `.class` se pueden ejecutar en distintas plataformas, como Windows, Linux, macOS, etcétera.

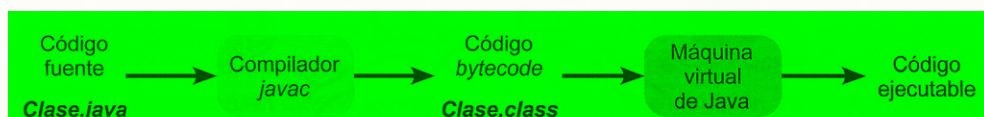


Figura 1.6. Proceso de obtención del código ejecutable de un programa escrito en Java: se parte del código fuente, que se plasma en un archivo con extensión .java por cada clase; con el compilador `javac` se transforma en un archivo con extensión `.class` escrito en código bytecode; finalmente, la JVM transforma este archivo en código ejecutable.

## 1.5. La ingeniería del software

La definición de software que propuso Pressman (véase Apartado 1.1) deja patente que el software no es solo un programa que funciona, que es obviamente lo más importante, sino que también comprende las estructuras de datos sobre las que opera el programa y la documentación resultado del proceso de desarrollo, así como la que describe cómo se debe usar el programa.

Es importante destacar que el proceso de desarrollo de software no fue concebido en sus inicios como un proceso de ingeniería, sino más bien como un trabajo artesanal en el que cada persona desarrolladora de software se ponía a programar directamente sin llevar a cabo fases previas y sin seguir unas normas mínimas.

En la década de los 70 del siglo xx se produjo lo que se conoce como la crisis del software, que se refiere a un conjunto de problemas encontrados en el desarrollo de software. Esta crisis abarcaba cómo desarrollar el software, cómo mantenerlo y cómo satisfacer la demanda creciente de software.

Los problemas fundamentales del software eran que:

- La planificación y estimación de costes eran frecuentemente muy imprecisas.

- La productividad de quienes desarrollaban el software no se correspondía con la demanda de sus servicios.
- La calidad del software no llegaba a veces a ser ni adecuada.

Se llegó a la conclusión de que todos estos problemas podían corregirse y que la clave estaba en dar un enfoque de ingeniería al desarrollo del software, junto con la mejora de las técnicas y las herramientas.

Todo esto desembocó en el nacimiento de la ingeniería del software, que es una disciplina para el desarrollo de software que surgió a partir de la ingeniería de sistemas y de hardware. La definición que propuso Fitz Bauer para la ingeniería del software es: «el establecimiento y uso de principios de ingeniería para llegar a un obtener un software rentable, que sea fiable y que funcione eficientemente en máquinas reales».



Figura 1.7. La ingeniería del software incluye una serie de tareas de desarrollo como la planificación, el análisis, diseño, programación, pruebas e implantación. Aquí también se muestran otros conceptos relacionados con el desarrollo de software, como el propio desarrollo y la validación y verificación de este.

El objetivo fundamental de la ingeniería del software es, por tanto, regular de alguna manera el desarrollo de software, es decir, establecer de manera clara las tareas que es necesario realizar para crear aplicaciones informáticas, realizando controles que aseguren que la calidad de los programas resultantes sea aceptable.

### Argot técnico

Según la RAE, el término *ingeniería* se refiere al «conjunto de conocimientos orientados a la invención y utilización de técnicas para el aprovechamiento de los recursos naturales o para la actividad industrial».

En el caso del software, esta definición es aplicable a la actividad industrial consistente en la creación de aplicaciones informáticas. Hace referencia, por tanto, al conjunto de conocimientos relacionados con el empleo de las técnicas necesarias para el desarrollo de software.



Se suele emplear también el término **proceso software** para referirse a los pasos necesarios para desarrollar una aplicación informática. En este sentido, una de las acepciones del vocablo **proceso** en el diccionario de la RAE es la siguiente: «conjunto de las fases sucesivas de un fenómeno natural o de una operación artificial». Se puede, por tanto, interpretar que el proceso de ingeniería del software o proceso software hace referencia al conjunto de fases para la construcción de software. En el Apartado 1.6, se estudiarán a fondo las tareas necesarias para el desarrollo de software.

## 1.6. Fases del desarrollo de una aplicación informática

Para la obtención de software, es necesario completar una serie de fases en las que se realizan una serie de tareas, que se exponen a continuación:

- **Análisis:** el **objetivo** de esta tarea es **analizar las necesidades de los usuarios potenciales del software para determinar qué debe hacer la aplicación y, de acuerdo con ello, escribir una especificación precisa de dicho sistema**. Durante esta fase se pretende responder a las siguientes preguntas: qué información ha de ser procesada, qué función y rendimiento se desean, qué interfaces deben establecerse, qué restricciones de diseño existen y qué criterios de validación se necesitan para definir un sistema correcto. Para ello, como es obvio, es necesario realizar una actividad inicial de comunicación con el cliente que ha encargado el desarrollo de la aplicación, y una vez obtenida la información necesaria, se podrá modelar el sistema. **Como resultado, se obtendrá una especificación del sistema, que consistirá en una documentación que describe lo que tiene que hacer el software, pero no cómo.** Es primordial que esta **documentación sea comprensible, completa y fácil de verificar y de modificar para facilitar todo el trabajo posterior**.
- **Diseño:** si bien el **objetivo** de la fase de análisis es determinar qué debe hacer el sistema, una vez se sabe esto, es preciso **establecer cómo se va a resolver el problema planteado**. Durante esta fase, **se traducen los requisitos** resultado de la fase de análisis **en componentes de software** (tablas de una base de datos, programas con procedimientos o funciones, clases con atributos y métodos, interfaz de usuario, etc.). Se trata de un refinamiento de la fase anterior. Por ejemplo, en esta fase, una de las decisiones que habrá que tomar es decidir qué **sistema gestor de base de datos** (SGBD) se va a emplear para gestionar la información que debe ser usada o generada por la aplicación: Oracle, MySQL, SQL Server, etcétera.
- **Programación:** consiste en **traducir los resultados de la fase de diseño en una forma legible para la máquina**, es decir, **se escribe el código fuente de cada componente del software empleando un determinado lenguaje de programación**. El **resultado** de esta etapa es el **código ejecutable**.
- **Pruebas:** con estas se **comprueba si la aplicación funciona correctamente**. Hay diferentes **tipos de pruebas**. Así, se **comienza probando cada componente de software por separado (pruebas unitarias)** y, posteriormente, se van integrando poco a poco los **componentes (pruebas de integración)**, hasta **probar el programa completo (pruebas**

de validación), cuyo objetivo es comprobar que la aplicación satisface los requisitos del cliente. Como resultado de estas pruebas, se descubrirán errores, en atención a los cuales, será necesario modificar el código, o incluso rehacer las tareas previas de diseño y/o análisis.

- **Ejecución:** en esta etapa, se lleva a cabo la instalación y puesta en marcha del software en el entorno de trabajo del cliente.
- **Mantenimiento:** el software sufrirá cambios después de que se entregue al cliente por diversos motivos. La tarea de mantenimiento consiste en realizar cambios sobre el software, para lo que se aplicará al programa existente, cada una de las actividades precedentes del ciclo de vida en vez de a uno nuevo.

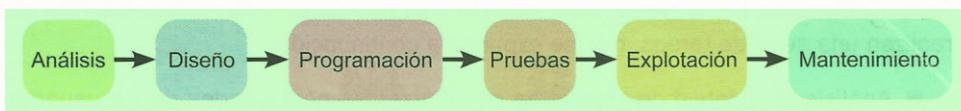


Figura 1.8. Fases del desarrollo de una aplicación informática.

### Recuerda



Aunque la tarea de mantenimiento parezca poco importante, hay estudios que indican que más del 50 % del esfuerzo de desarrollo de software se dedica a esta tarea.

Esto se debe a que es muy común que toda aplicación informática, a lo largo de su vida, debe ser sometida a ampliaciones, correcciones o a adaptaciones a nuevos entornos. Por este motivo, es muy importante tener presente que, al desarrollar software, lo más probable es que otras personas tengan que revisarlo y modificarlo, y por este motivo, nuestro trabajo siempre debe poder ser entendido por esas personas.

Además de las tareas técnicas que se han señalado, es necesario llevar a cabo otra tarea, que debe comenzar tras la comunicación inicial con el cliente y que es una actividad más de gestión que técnica. Se trata de la tarea de planificación, que consiste en elaborar un plan del proyecto de software que indique las tareas técnicas que se han de realizar, los riesgos probables, los recursos que se necesitan, los productos de trabajo que se obtendrán y una programación de las actividades (qué actividades se deben ejecutar, en qué orden, la duración de cada actividad, los recursos humanos y materiales necesarios para cada actividad, etc.).

Por otro lado, además de todas estas actividades, que Pressman (2010) denomina actividades estructurales, para el desarrollo de software, es preciso realizar otras tareas adicionales, de menor carácter técnico y también relacionadas con la gestión, y que Pressman (2010) llamó actividades sombrilla:

- **Seguimiento y control del proyecto de software:** se debe evaluar periódicamente el proyecto comparándolo con el plan preestablecido con el fin de tomar acciones si se produce algún desvío respecto al plan.
- **Administración del riesgo:** consiste en evaluar los riesgos (problemas) que pueden afectar al resultado del proyecto o a la calidad del producto.

- **Aseguramiento de la calidad del software:** se definen y ejecutan las actividades necesarias para garantizar la calidad del software.
- **Revisiones técnicas:** se evalúan los productos obtenidos con el fin de descubrir y eliminar errores antes de que se propaguen a la siguiente actividad.
- **Medición:** consiste en definir y reunir las mediciones del proceso, proyecto y producto para ayudar al equipo a entregar el software que satisfaga las necesidades del cliente.
- **Administración de la configuración del software:** se administran los efectos del cambio a lo largo del proceso de desarrollo de software.
- **Administración de la reutilización:** se establecen los criterios para poder usar productos del trabajo en proyectos posteriores y los mecanismos para que esto se pueda llevar a la práctica, es decir, para obtener componentes reutilizables.
- **Preparación y producción del producto del trabajo:** se agrupan las actividades necesarias para crear productos del trabajo, como modelos, documentos, registros, formatos, etcétera.

Seguidamente, se amplía la explicación de cada una de las actividades estructurales de la ingeniería del software.

## ■ ■ ■ 1.6.1. Análisis

El **objetivo** de esta actividad estructural de análisis es **lograr una comprensión clara del problema que se necesita resolver o, dicho de otro modo, de los requisitos funcionales que debe cumplir la aplicación que el cliente ha encargado.**

Esta tarea no es sencilla por diversos motivos:

- El cliente puede no tener claro qué requisitos debe satisfacer la aplicación.
- El cliente tiene claros los requisitos, pero no es capaz de expresarlos de manera comprensible para el equipo de desarrollo.
- Puede haber malentendidos entre el equipo de desarrollo y el cliente por la razón indicada en el punto anterior, por desconocimiento del problema que hay que resolver por parte del equipo de desarrollo, o por desconocimiento de lo que la tecnología puede ofrecer por parte del cliente.

Además de los requisitos funcionales, que hacen referencia a lo que tiene que hacer la aplicación, hay otro tipo de **requisitos** llamados **no funcionales**, como los siguientes:

- **Fiabilidad:** grado en que una aplicación funciona sin fallos.
- **Escalabilidad:** capacidad del sistema para manejar aumentos de carga sin disminuir el rendimiento.
- **Extensibilidad:** capacidad del software para añadir nuevas funcionalidades y componentes.

- **Seguridad:** grado con que una aplicación protege la información contra el acceso indebidamente a ella.
- **Mantenibilidad:** grado en que el software es comprendido, reparado o mejorado.

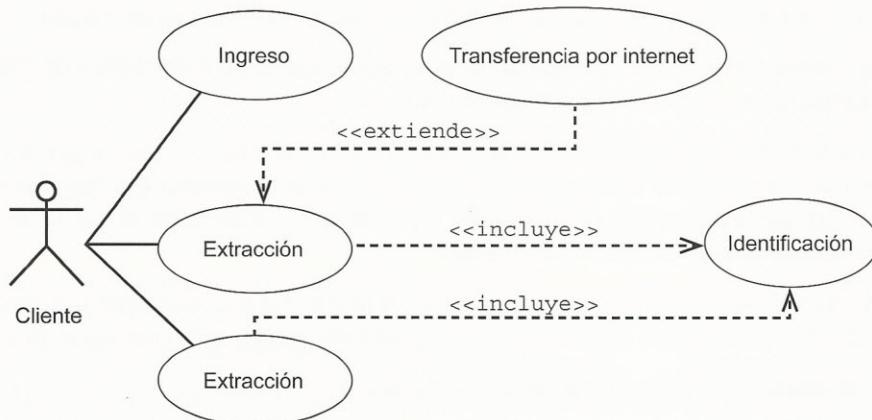
Estos requisitos también hay que tenerlos en cuenta para crear una aplicación que responda a las necesidades del cliente.

Antes de elaborar los modelos que son el resultado de la fase de análisis, es necesaria una tarea de **comunicación con el cliente**, para la que se pueden emplear **distintas técnicas**, como:

- **Entrevistas:** es la técnica más tradicional. Es similar a una entrevista periodística en la que el analista entrevista, uno a uno, a los futuros usuarios del software.
- **Desarrollo conjunto de aplicaciones (JAD)**, por sus siglas en inglés, *joint application development*): se crea un equipo de analistas y usuarios que se reúnen para trabajar conjuntamente en la determinación de las necesidades de los usuarios.
- **Desarrollo de un prototipo:** consiste en construir un modelo o maqueta del sistema que permite ver a los usuarios las características del sistema que desean obtener. El prototipo se puede o bien usar solo para este fin y se desecha, o bien se puede mejorar para convertirlo en el producto final.

Como resultado de esta fase, se debe obtener un documento llamado **especificación de requisitos del software (ERS)**, que sirve de base para la siguiente etapa de diseño.

En la ERS, se incluyen **modelos gráficos con apoyos textuales**. Dichos modelos tienen la ventaja de no ser ambiguos, a diferencia del lenguaje natural (español, inglés, etc.), y al ser creados con herramientas informáticas, son fácilmente modificables. En esta etapa, se crearán tanto **diagramas de clases** como **diagramas de casos de uso**, que se estudiarán en las Unidades 5 y 6 de este libro, respectivamente. En la Figura 1.9, se muestra un diagrama de casos de uso.



**Figura 1.9.** Este diagrama de casos de uso refleja los requisitos funcionales que debe satisfacer una aplicación para una entidad bancaria en la que los usuarios pueden realizar tres tipos de operaciones con su cuenta bancaria: ingresar dinero, extraer dinero o realizar una transferencia. Para los dos últimos tipos de operaciones, los clientes se tienen que identificar.

## ■■■ 1.6.2. Diseño

Partiendo de la ERS obtenida en la etapa anterior, que indica, en esencia, cuál es el problema que se ha de resolver con la aplicación, **en la etapa de diseño, se determina cómo se ha de solucionar dicho problema**. Para ello, **se partirá de los diagramas obtenidos en la etapa de análisis**, refinando algunos de ellos y creando otros que indiquen los pasos que hay que dar para responder a los requisitos establecidos.

Por ejemplo, partiendo del diagrama de casos de uso obtenido en la etapa de análisis, se tratará de establecer qué acciones hay que llevar a cabo para ejecutar cada caso de uso, teniendo en cuenta que cada caso de uso se corresponde generalmente con un requisito funcional. Así, tomando como ejemplo el diagrama de casos de uso de la Figura 1.9, en la etapa de diseño, habrá que indicar cuáles son los pasos necesarios para realizar una transferencia de dinero de una cuenta a otra. De esta forma, cada caso de uso genera un diagrama de interacción que detalla los pasos necesarios para ejecutarlo.

## ■■■ 1.6.3. Programación

Se parte del diseño detallado y, en esta fase, **se escribe el código fuente correspondiente siguiendo las normas del lenguaje de programación de alto nivel seleccionado**.

Es habitual que se establezcan determinadas normas a la hora de escribir el código fuente con el fin de que los programas tengan un aspecto homogéneo y sean fáciles de entender y, por tanto, de modificar, sobre todo si se tiene en cuenta la relevancia de las posteriores tareas de mantenimiento. Normalmente, se establecen normas en relación con los siguientes elementos:

- **Comentarios:** se suelen escribir al comienzo de determinadas partes del código para describir su razón de ser (al comienzo de una clase, al indicar cada atributo de una clase, al comienzo de cada método, etc.) y para aclarar determinadas porciones de código por su alto nivel de complejidad.
- **Declaraciones de variables y de parámetros de métodos.**
- **Nombres de clases, atributos, métodos, variables, constantes, etc.**
- **Líneas en blanco entre clases, métodos, etc.**
- **Sangrados para facilitar la legibilidad del código.**

A modo de ejemplo, se muestran dos representaciones del código fuente de un programa que suma los números pares comprendidos entre el 1 y el número introducido por el teclado. La segunda versión del programa es más comprensible porque incluye comentarios, sangrados y líneas en blanco para separar los elementos del código.

*Primera versión del programa:*

```
1 package programas; import java.util.Scanner;  
2 public class PruebaSumaPares {
```

```

3 public static void main(String[] args) {
4     int suma = 0; int num = 1;
5     System.out.print ("Introduzca un número mayor o igual que 1: ");
6     Scanner teclado = new Scanner (System.in);
7     int máximo = teclado.nextInt();
8     while (num <= máximo){
9         if (num % 2 == 0) suma = suma + num; ++num;
10        System.out.println ("La suma de los números pares entre 1 y "
11                           + máximo + " es "+suma);
12    }
13 }
```

Segunda versión del programa:

```

1 package programas;
2 import java.util.Scanner;
3 public class PruebaSumaPares {
4     public static void main(String[] args) {
5         //Declaramos una variable para que contenga el resultado de
6         // las sumas
7         int suma = 0;
8         /*Usamos la variable num para contener todos los números
9         comprendidos entre 1 y máximo*/
10        int num = 1;
11        System.out.print ("Introduzca un número mayor o igual que 1: ");
12        //Leemos un número entero por teclado y lo almacenamos en
13        //máximo
14        Scanner teclado = new Scanner (System.in);
15        int máximo = teclado.nextInt();
16        //Sumamos los números pares comprendidos entre 1 y máximo
17        while (num <= máximo){
18            if (num % 2 == 0)
19                suma = suma + num;
20            ++num;
21        }
22        System.out.println ("La suma de los números pares entre 1 y "
23                           + máximo + " es "+suma);
24    }
25 }
```

## ■■■ 1.6.4. Pruebas

El objetivo de esta actividad es detectar errores en el software antes de que este sea entregado al cliente. La estrategia que se aplica en dichas pruebas consiste en realizarlas desde los elementos más pequeños hasta los más grandes, es decir, se comienza probando pequeños componentes de software hasta llegar al programa completo.

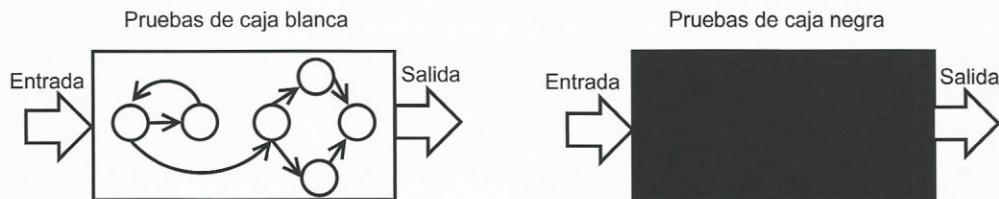
El objetivo de las pruebas es doble: **verificar y validar el software**. En otras palabras:

- Por medio de la **verificación**, se determina si el software se ha construido correctamente, es decir, si las tareas que realiza las lleva a cabo de manera adecuada.
- La **validación** consiste en comprobar si el software es realmente el que el usuario desea, o sea, si el sistema hace lo que quiere el usuario.

Existen dos tipos de técnicas de pruebas:

1. **Las pruebas de caja blanca o pruebas estructurales:** se examinan los detalles de cada módulo, para lo que se debe disponer del código fuente. Así, se prueban los diferentes caminos a través de este código, así como los bucles, las variables, etcétera.
2. **Las pruebas de caja negra o pruebas funcionales:** el software se considera como una caja negra que recibe una serie de entradas y proporciona una serie de salidas. El objetivo es validar los requisitos funcionales.

Lo habitual es emplear estas dos técnicas de pruebas. De hecho, se pueden combinar y se complementan perfectamente. Si al realizar una prueba esta tiene éxito, es decir, si se detecta algún error, se procede al proceso de depuración, que consiste en localizar y corregir el error. Esto puede suponer realizar cambios solo en la programación, o también en etapas previas, como el diseño, o incluso el análisis.



**Figura 1.10.** En las pruebas de caja blanca es necesario conocer el código fuente para realizar las pruebas, mientras que en las de caja negra, solo es necesario conocer las funciones que realiza el software para comprobar si la salida obtenida coincide con la esperada.

### 1.6.5. Explotación

Una vez que se ha probado el software, se han corregido los errores detectados y se ha documentado todo el proceso, se lleva a cabo la instalación y la puesta a punto y en funcionamiento de la aplicación en las instalaciones del cliente. Durante esta fase, es **necesaria la presencia del cliente**. Por lo tanto, las **tareas de esta fase** son:

- Se instalan los programas en el equipo o los equipos del cliente.
- Se llevan a cabo las **comprobaciones conocidas como pruebas beta**, que se realizan en las instalaciones del cliente.
- Se realiza la **configuración del sistema** y se comprueba que la aplicación funciona de manera adecuada.
- Una vez completados los pasos anteriores, la aplicación ya pasa a la fase de **producción normal**, esto es, a su uso por parte del usuario final.

## ■ ■ ■ 1.6.6. Mantenimiento

Hay diferentes tipos de mantenimiento en función del motivo que lo origina:

- **Mantenimiento correctivo:** este se debe a que el cliente detecta errores en el software a pesar de las pruebas que se realizaron, y consiste en corregir estos errores.
- **Mantenimiento adaptativo:** en muchos casos el software deba adaptarse a cambios del entorno externo porque, por ejemplo, se tiene un nuevo sistema operativo. Este tipo de mantenimiento se realiza frecuentemente por los rápidos avances en el ámbito de la informática.
- **Mantenimiento perfectivo:** es muy común que, con el paso del tiempo, el cliente solicite ampliaciones funcionales, es decir, desee que la aplicación incorpore nuevos requisitos funcionales. Asimismo, es posible que se deseen mejoras en requisitos no funcionales, por ejemplo, que el programa se ejecute más rápidamente o que se incremente la seguridad del sistema.

## ■ 1.7. Roles que intervienen en el proceso de desarrollo de software

Puesto que el desarrollo de software se puede dividir en distintas tareas, es habitual contar con una persona experta para cada una de ellas, lo que origina que en el campo de la ingeniería del software existan diferentes roles:

- **Jefe de proyecto:** lleva a cabo la tarea de planificación del proyecto, que incluye, además de la propia planificación, la puesta en marcha del proyecto, su ejecución, seguimiento, control y cierre. Se trata, pues, de una tarea más de gestión que técnica pero de gran relevancia, pues puede influir de manera decisiva en el éxito o fracaso del proyecto. Se trata del máximo responsable de que el proyecto salga adelante.
- **Expertos del dominio:** son personas que trabajan en la empresa u organismo para el que hay que desarrollar la aplicación y que conocen el dominio del problema, es decir, la parte de la realidad para la que se va a crear la aplicación. Los técnicos deben comunicarse con estas personas, puesto que son las que conocen en detalle los requisitos que la aplicación debe satisfacer.
- **Analista:** su cometido es la tarea de análisis, que consiste en crear un modelo claro y consistente que se corresponda con la visión del problema que ha conseguido con la ayuda de los expertos del dominio. Este rol también se conoce frecuentemente como **analista funcional**.
- **Arquitecto:** a partir del trabajo realizado por el analista, debe definir las líneas maestras del diseño, estableciendo la arquitectura del sistema, esto es, la manera en que este se divide en una serie de componentes. Esta arquitectura deberá ser tomada como referencia por el resto de personas desarrolladoras para llevar a buen término su trabajo.

- **Diseñador:** parte de un subconjunto de los requisitos definidos por el analista y de la arquitectura del sistema para **diseñar las partes del sistema que implementan esos requisitos hasta un nivel de detalle suficiente para acometer la tarea de programación**. Este rol también se conoce como **analista orgánico** o **analista técnico**.
- **Programador:** se **encarga de la tarea de programación**, es decir, de **escribir el código fuente partiendo del diseño detallado realizado por el diseñador**.
- **Probador:** se encarga de **la tarea de pruebas** y, por tanto, de **garantizar la calidad de la aplicación creada**. Para ello partirá de los requisitos y de los programas, y se encargará de probarlos para decidir, en último término, si están en condiciones de ser entregados al usuario final.
- **Encargado de la implantación:** es el **encargado de que se realice el empaquetado de los programas y su instalación en el entorno de trabajo del cliente**. También es el **encargado de gestionar la configuración** y, por tanto, las diferentes versiones que vayan surgiendo de la aplicación a partir de los cambios que se vayan incorporando.

Esta distribución de roles no indica necesariamente que cada rol sea desempeñado por una sola persona ni tampoco que una misma persona no pueda desempeñar varios roles. Estos hacen referencia simplemente a los distintos papeles que deben desempeñar personas en el desarrollo de software, pero cada uno puede ser desempeñado por una o varias personas o, incluso, una persona puede desempeñar varios de estos roles, lo que dependerá de las características de la aplicación que se esté desarrollando, del equipo de desarrollo con que se cuente y de las características y organización de la empresa de desarrollo.



Figura 1.11. El trabajo conjunto y cohesionado de todos los miembros del equipo de desarrollo bajo el mando de un líder (jefe de proyecto) es un aspecto fundamental para el éxito del proyecto.

## ■ 1.8. Modelos de ciclo de vida del software

Si bien las actividades necesarias para el desarrollo de software son las expuestas en el Apartado 1.6, estas se pueden organizar de diferentes maneras en función de las características del proyecto o del producto, del personal involucrado en el desarrollo, etc. Esto se plasma en el empleo de un determinado ciclo de vida del software.

Piattini et al. (2007) proporcionan la siguiente definición para el **concepto de ciclo de vida**: «el ciclo de vida del software es la descripción de las distintas formas de desarrollo de un proyecto o aplicación informática, es decir, la orientación que debe seguirse para obtener, a partir de los requerimientos del cliente, sistemas que puedan ser utilizados por dicho cliente».

Un ciclo de vida **determina el orden en el que se deben llevar a cabo las tareas en el proceso de desarrollo de software y los criterios que se deben cumplir para realizar el paso de una tarea a la siguiente**. Estos criterios consisten en la obtención de ciertos productos intermedios y, también, en indicar las características que deben satisfacer estos productos para progresar a la fase posterior. Por ejemplo, como resultado de la tarea de análisis, se debe obtener una ERS y se puede establecer qué condiciones debe cumplir esta para poder avanzar a la fase de diseño.

De todo ello se deduce que la selección del ciclo de vida para un proyecto **es una tarea muy relevante que puede influir de manera decisiva en su éxito o fracaso**.

El ciclo de vida que surgió en primer lugar es el llamado ciclo de vida clásico o **modelo en cascada**. Posteriormente, se desarrollaron otros modelos: **de proceso incremental, de proceso evolutivo y de desarrollo ágil**. En los siguientes apartados, se describe cada uno de ellos.

### ■ 1.8.1. Modelo en cascada

El modelo en cascada, también llamado **ciclo de vida clásico**, es el modelo más primitivo de ciclo de vida, pero ha resultado fundamental para el progreso posterior porque en él se identifican ya prácticamente todas las clases de actividades distintas que intervienen en el desarrollo y explotación de software.

En este modelo, se **propone un enfoque sistemático y secuencial para el desarrollo de software, comenzando con el análisis y continuando a través del diseño, la programación y las pruebas hasta llegar al software terminado, sobre el que se pueden aplicar también operaciones de mantenimiento**. La Figura 1.12 muestra este flujo de actividades.

Se plantea un flujo secuencial de las actividades y para pasar de una fase a la siguiente es necesario cumplir ciertos objetivos. Para detectar los errores lo antes posible y evitar que se propaguen a fases posteriores, se establece un proceso de revisión al completar cada fase, antes de pasar a la siguiente. Esta revisión se realiza fundamentalmente sobre la documentación producida en esa fase y se hace de una manera formal. Si, a pesar de todo, durante la realización de una fase, se detectan errores en el resultado de fases anteriores, será necesario rehacer parte del trabajo volviendo a un punto anterior

del ciclo de vida, como se indica con las flechas de la Figura 1.12. Por ejemplo, si en la etapa de mantenimiento, se detecta que se cometió algún error en el diseño, será necesario corregir dicho error de diseño y realizar cambios en las etapas posteriores de programación y pruebas.

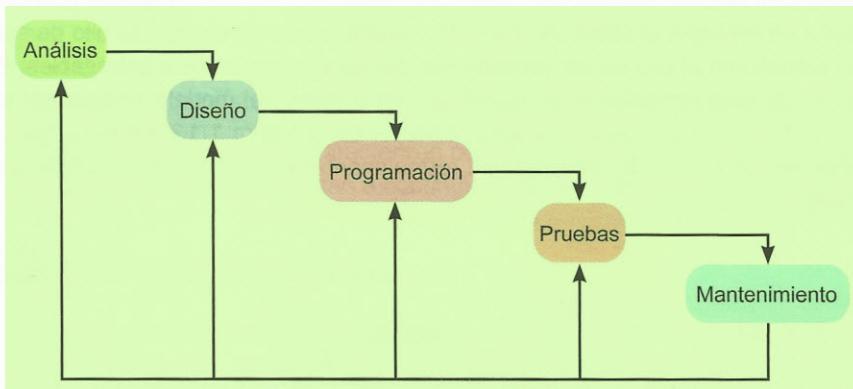


Figura 1.12. Representación del modelo en cascada o ciclo de vida clásico. Las cinco actividades que se muestran se deben ejecutar en secuencia, si bien se permite volver hacia atrás cuando se detecta la necesidad de realizar algún cambio.

Este modelo tiene, entre otras **ventajas**, que es **fácil de comprender, de planificar y de seguir, y existen además herramientas que lo soportan**. Sin embargo, entre los **problemas** de este modelo se encuentran los siguientes (Pressman, 2010):

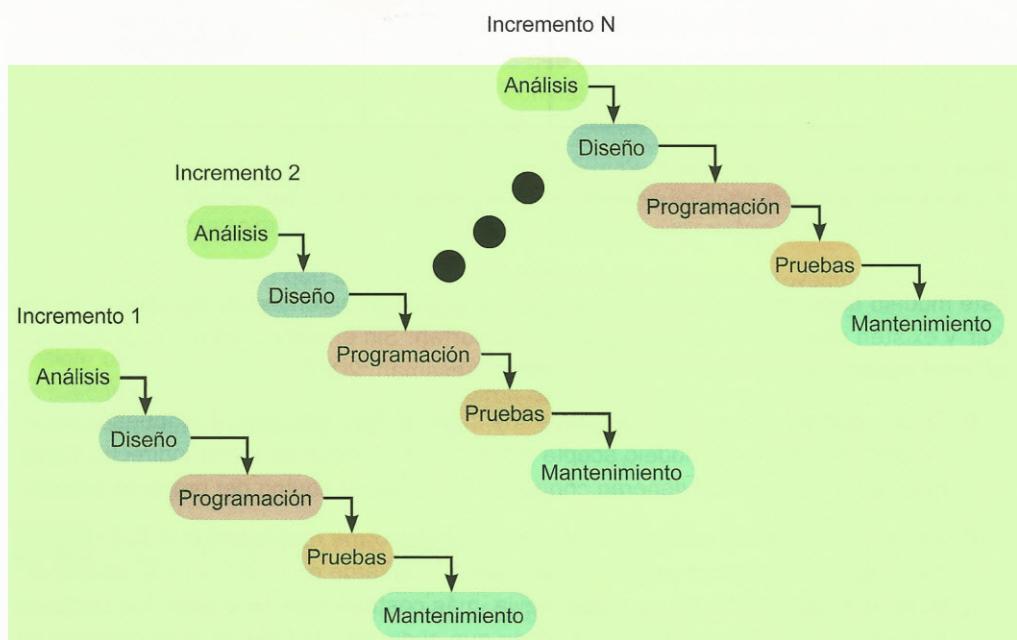
- En la realidad, **es raro que los proyectos sigan el flujo secuencial propuesto por el modelo**. Aunque este modelo acepta repeticiones, lo hace de forma indirecta. Como resultado, los cambios generan confusión conforme el equipo del proyecto avanza.
- A menudo, **es difícil para el cliente enunciar de forma explícita todos los requerimientos**. Se debe tener presente que cuanto más tarde en el proceso de desarrollo se detecte algún fallo en una fase previa, más costoso será incorporar los cambios que ese fallo implique. Esto quiere decir que si durante la fase de programación se ha detectado que algún requisito no se ha incorporado o se ha hecho de manera errónea, será necesario rehacer todas las fases previas del ciclo de vida (análisis y diseño).
- **El cliente debe tener paciencia, ya que no se dispondrá de una versión funcional del programa hasta que el proyecto esté muy avanzado**. Sería desastroso si al revisar el funcionamiento del programa supuestamente terminado, se detectara un fallo grande en este.

Hoy día, el desarrollo de software está sometido a multitud de cambios, motivo por el cual este modelo de ciclo de vida no resulta apropiado en la mayoría de los casos. A pesar de ello, se podría emplear en aquellos casos en los que no es previsible que se produzcan cambios en los requisitos.

Dadas las desventajas de este modelo, se desarrollaron con posterioridad otros modelos que se exponen a continuación.

## 1.8.2. Modelos de proceso incremental

Uno de los inconvenientes más importantes del modelo en cascada es que el cliente no dispone de una versión funcional del sistema hasta muy tarde. En muchos casos puede resultar más conveniente proporcionar cierta funcionalidad parcial del software al cliente y aumentarla en entregas posteriores. De esta manera, se concibe el desarrollo de software como un proceso en el que se van produciendo diversos incrementos o entregables. Para la elaboración de cada incremento, se siguen las actividades del modelo en cascada que se indicaron anteriormente, como se puede observar en la Figura 1.13. En esta figura, para simplificar, se han eliminado las líneas de retroalimentación que van de una fase a las fases previas.



**Figura 1.13. Representación del modelo incremental de desarrollo de software.** El software se divide en una serie de incrementos, en cada uno de los cuales se aplican los pasos del ciclo de vida clásico.

Entre las **ventajas** de este modelo, Cuevas et al. (2003) señalan las siguientes:

- Reduce los riesgos de retrasos, de cambios de requisitos y problemas de aceptación.
- Los entregables intermedios facilitan la retroalimentación por parte de los clientes, lo cual resulta ventajoso para evitar problemas mayores en los siguientes entregables.
- Permiten al usuario validar el sistema a medida que se construye.

Resulta recomendable emplear este modelo de ciclo de vida cuando no están bien definidos los requisitos y es muy probable que se produzcan cambios.

### 1.8.3. Modelos de proceso evolutivo

Los modelos de proceso evolutivo tienen su razón de ser en el hecho de que «el software, como todos los sistemas complejos, evoluciona en el tiempo». Es frecuente que los requerimientos del negocio y del producto cambien conforme avanza el desarrollo, lo que hace que no sea realista trazar una trayectoria rectilínea hacia el producto final [...] Se comprende bien el conjunto de requerimientos o el producto básico, pero los detalles del producto o extensiones del sistema aún están por definirse» (Pressman, 2010).

Estos modelos son iterativos y se caracterizan por el desarrollo de versiones cada vez más completas del software. Se presentan a continuación dos modelos de proceso evolutivo:

#### Construcción de prototipos

Un **prototipo** se puede definir como un sistema auxiliar que permite probar experimentalmente ciertas soluciones parciales a las necesidades del usuario o a los requisitos del sistema. Puede tratarse de un modelo que describa la interacción hombre-máquina, de manera que facilite al usuario la comprensión de cómo se producirá la interacción, o un programa que implemente algunos subconjuntos de la funcionalidad de la aplicación. Es conveniente seguir este modelo de ciclo de vida cuando el cliente es capaz de definir un conjunto de objetivos generales para el software, pero no puede identificar los requerimientos en detalle, o cuando las personas desarrolladoras tienen dudas considerables sobre determinados aspectos importantes del desarrollo, como la eficiencia de un algoritmo, en qué medida la aplicación se podrá usar en un determinado sistema operativo o de qué forma se ha de llevar a cabo la interacción hombre-máquina. Se puede considerar este modelo de ciclo de vida de manera aislada, pero es más común emplearlo como una técnica que puede implementarse en el contexto de cualquiera de los modelos de ciclo de vida descritos en este tema.

Hay que tener en cuenta que el **objetivo fundamental** a la hora de desarrollar el prototipo es que las y los desarrolladores sepan exactamente lo que tiene que hacer la aplicación. En la Figura 1.14, se presenta la secuencia de tareas que se llevan a cabo según este modelo. Se comienza con una actividad de comunicación con el cliente con el fin de definir los objetivos generales del software y detectar las áreas en las que es necesaria una mayor definición. Luego se planifica una iteración para desarrollar el prototipo y se lleva a cabo el modelado (diseño rápido). Seguidamente, se construye el prototipo, que se entrega y es evaluado por los participantes, quienes proporcionan retroalimentación a las personas desarrolladoras con el fin de mejorar la respuesta a los requerimientos. La iteración ocurre a medida que el prototipo es refinado para satisfacer las necesidades de distintos participantes y, al mismo tiempo, le permite a las y los desarrolladores entender mejor lo que se necesita hacer.

En este modelo, se construye el software a partir de versiones cada vez más completas, de manera que se realizan una serie de tareas varias veces. En la primera iteración, se llevan a cabo las tareas de la Figura 1.14 (planificación, modelado, construcción, despliegue, evaluación y comunicación) para construir una primera versión del prototipo. La reactualización de todas estas actividades constituye una iteración, luego se vuelven a realizar

estas mismas tareas en la segunda iteración para obtener una segunda versión del prototipo más completa, y así sucesivamente.

Hay que tener en cuenta que **algunos prototipos son desecharables**, mientras que **otros son evolutivos**, es decir, se transforman poco a poco hasta convertirse en el sistema real.



Figura 1.14. Actividades del modelo de construcción de prototipos.

### ■ ■ ■ Modelo en espiral

Este **modelo evolutivo** fue diseñado para cubrir las mejores características tanto del **modelo en cascada** como del **modelo de construcción de prototipos**, añadiendo un nuevo elemento, que es el análisis de riesgo, dentro de la actividad estructural de planificación. Como modelo de desarrollo evolutivo que es, el software se desarrolla en una serie de entregas que se pueden representar como iteraciones o vueltas alrededor de una espiral. Durante las primeras iteraciones lo que se entrega puede ser un modelo o prototipo. En las iteraciones posteriores se producen versiones cada vez más completas del software. Se muestra una representación de este modelo en la Figura 1.15.



Figura 1.15. Representación del modelo en espiral, indicando las actividades que se llevan a cabo en cada cuadrante.

Se describe a continuación el cometido de las tareas que se llevan a cabo en cada ciclo alrededor de la espiral:

- **Planificación:** se identifican los objetivos que se desean conseguir mediante la iteración, así como las alternativas para la consecución de dichos objetivos y las restricciones impuestas en cuanto a plazos, costes, etc.
- **Análisis de riesgo:** consiste en evaluar las diferentes alternativas para la realización de la parte del desarrollo elegida, seleccionando la más ventajosa y tomando precauciones para evitar los inconvenientes o riesgos previstos.
- **Ingeniería:** se desarrolla el producto (análisis, diseño, programación, etc.). El objetivo es ir obteniendo en cada ciclo una versión más completa del sistema.
- **Evaluación:** el resultado del trabajo de ingeniería es evaluado por el cliente y se decide si se continúa, para lo que se procede a planificar la siguiente iteración.

La principal ventaja de este modelo es que la consideración de los riesgos o problemas que pueden afectar al desarrollo del software en diferentes momentos permite que tanto quienes encargaron el software como los que lo desarrollan sean conscientes de los problemas que pueden presentarse y sean capaces de reaccionar en caso de que estos ocurran. Además, si hay cierta inseguridad en cuanto a sus requisitos, se puede aplicar el enfoque de construcción de prototipos para eliminar ese riesgo.

El principal inconveniente de este modelo es, como indica Pressman (2010), que «requiere mucha experiencia en la evaluación del riesgo y se basa en ella para llegar al éxito. No hay duda de que habrá problemas si algún riesgo importante no se descubre y administra».

## Recuerda



Los dos modelos de proceso evolutivo que se han presentado (construcción de prototipos y modelo en espiral) tienen dos características principales en común:

- La consideración del proceso de desarrollo de software como un proceso incremental en el que se van obteniendo versiones del software cada vez más completas hasta llegar al producto final.
- La combinación del modelo evolutivo correspondiente con el modelo en cascada, que en lugar de aplicarse sobre todo el software, se aplica sobre incrementos de software o versiones de la aplicación cada vez más completas. De esta manera, se saca provecho de las mejores características de varios modelos de ciclo de vida.

## Actividad resuelta 1.1

### El modelo en espiral y el modelo en cascada

Describe en detalle de qué manera se combinan y complementan el modelo en cascada y el modelo en espiral.

### Solución

Cuando se sigue el modelo en espiral, al tratarse de un modelo de proceso evolutivo, se va construyendo el software sobre la base de incrementos o versiones del software cada vez más completas. **El conjunto de tareas por medio de las cuales se construye cada uno de dichos incrementos recibe el nombre de iteración.** En cada iteración en el cuadrante de ingeniería, se llevan a cabo las tareas propias del modelo en cascada antes de la entrega de la aplicación (análisis, diseño, programación y pruebas). Por tanto, se aplica el modelo en cascada, pero no sobre el producto final, sino sobre incrementos de software cada vez más completos o versiones de la aplicación que incorporan progresivamente más funcionalidad. Una vez que se ha creado cada uno de estos incrementos, se muestra al cliente, quien lo evalúa, y se pasa a la siguiente iteración, hasta que, como consecuencia de la última iteración, se entrega el software completo y se procede a su implantación final en las instalaciones del cliente (explotación).

### Actividad propuesta 1.3

#### El modelo en espiral y el modelo de construcción de prototipos

Explica detalladamente de qué forma se combinan y complementan el modelo en espiral y el modelo de construcción de prototipos.

## ■ 1.9. Metodologías de desarrollo de software

Maddison (1983) propone la siguiente definición para el **concepto de metodología**: «**un conjunto de filosofías, fases, procedimientos, reglas, técnicas, herramientas, documentación y aspectos de formación para los desarrolladores de sistemas de información**». Una metodología puede seguir uno o varios modelos de ciclo de vida, esto es, el ciclo de vida indica qué es lo que hay obtener a lo largo del desarrollo del proyecto, pero no cómo, que es lo que indicará la metodología. Por tanto, el ciclo de vida es el modelo general que se sigue para el desarrollo de software, mientras que **una metodología establece las tareas concretas que hay que llevar a cabo, cómo se debe llevar a cabo cada tarea, con qué herramientas, etcétera**.

Es necesario detenerse aquí y aclarar primero los conceptos de *procedimiento*, *técnica* y *herramienta* que aparecen en la definición anterior de metodología, junto con los conceptos de *tarea*, *producto intermedio* y *producto final*:

1. **El proceso de desarrollo de software, para que sea manejable, ha de dividirse en una serie de fases en las que se deben llevar a cabo una serie de tareas.**
2. **Para llevar a cabo cada tarea, se debe seguir un determinado procedimiento y, como consecuencia de su realización, se obtienen uno o varios productos, los cuales pueden ser intermedios, si se usan como base para la realización de alguna tarea posterior, o finales, en caso de que se entreguen al cliente al final del proceso de desarrollo.**

3. Para aplicar cada procedimiento, se deben utilizar uno o varios métodos o técnicas, en las que se crean diagramas gráficos con apoyos textuales. Además, para aplicar cada método se puede disponer de la ayuda de alguna herramienta que automatice en mayor o menor medida la aplicación de dicho método. Un mismo método se puede usar varias veces en una misma metodología y también se puede utilizar en diferentes metodologías.

Las herramientas que permiten automatizar las tareas de desarrollo de software reciben el nombre de herramientas CASE (del inglés, computer-aided software engineering).

De las definiciones de método y metodología, se puede deducir pues que una metodología incluye un conjunto de métodos para realizar cada una de las tareas necesarias para el desarrollo de software.

Las metodologías de desarrollo de software han ido evolucionando a lo largo del tiempo. Así, se identifican tres períodos de tiempo:

1. **Desarrollo convencional:** durante los primeros años del desarrollo de software las prácticas de desarrollo eran totalmente artesanales y no se seguía ninguna metodología, lo que acarreaba multitud de problemas y desembocó en lo que se llamó la crisis del software.
2. **Metodologías estructuradas:** la primera respuesta a esta crisis fue la regulación de la tarea de programación con la difusión de la programación estructurada, a la que siguió el surgimiento de métodos para el diseño y análisis estructurado, dando lugar a las metodologías estructuradas, que abarcaban la totalidad del ciclo de vida del software.
3. **Metodologías orientadas a objetos:** en la década de los 80 del siglo pasado surgieron, en primer lugar, los lenguajes orientados a objetos y más tarde, métodos para el diseño y análisis orientados a objetos.

El paradigma orientado a objetos supuso un cambio de filosofía en relación con la manera de desarrollar software que se había empleado hasta entonces, la cual se basaba en la metodología estructurada.

Como se recordará por lo indicado en el Apartado 1.2, en la metodología estructurada, las aplicaciones se conciben como programas compuestos por diversos componentes de software llamados módulos entre los que se realizan llamadas. Sin embargo, en la metodología orientada a objetos, la atención se centró más que en los procesos, en los datos sobre los que hay que operar. Una aplicación consta de varios objetos, cada uno de los cuales tiene un estado (definido por un conjunto de atributos) y un comportamiento (definido por una serie de operaciones o métodos que ejecuta al recibir un mensaje de otro objeto).

En la actualidad, la metodología que está sirviendo como referencia es el denominado proceso unificado de desarrollo de software de Rational Software Corporation, que está avalado por Ivar Jacobson, Grady Booch y James Rumbaugh, autores del lenguaje unificado de modelado (UML, por sus siglas en inglés, unified modeling language). En el Apartado 1.9.1, se estudia a fondo esta metodología.



## Argot técnico

El lenguaje UML es muy conocido y, aunque en su nombre se incluye el vocablo *lenguaje*, este lenguaje es un tanto especial en el sentido de que **se trata de un lenguaje gráfico que establece un conjunto de normas que permiten visualizar, construir, especificar y documentar un sistema**. Siguiendo estas normas, se pueden dibujar distintos tipos de diagramas que representan diferentes aspectos de un sistema siguiendo el paradigma orientado a objetos.

Así, en la Unidad 5, se estudian las normas de UML para el dibujo de diagramas de clases y, en la Unidad 6, se explica cómo dibujar los diferentes tipos de diagramas que reflejan el comportamiento de un sistema, como los diagramas de casos de uso y los diagramas de interacción.

Recientemente, se han desarrollado metodologías de desarrollo novedosas conocidas como **modelos de desarrollo ágil**, que **intentan simplificar las metodologías existentes**, buscando un equilibrio entre seguir un proceso de desarrollo que sea excesivo o «muy burocrático» y su inexistencia. Estas metodologías se analizan en el Apartado 1.9.2.

### 1.9.1. El proceso unificado de Rational (RUP)

El proceso RUP (Kruchten, 2000) **se puede describir en función de dos dimensiones**:

- **Dimensión temporal:** se expresa en términos de ciclos, fases, iteraciones e hitos.
- **Dimensión estática:** se expresa en términos de actividades (*activities*), productos intermedios (*artifacts*), perfiles de trabajo o roles (*workers*) y flujos de trabajo (*workflow*).

Atendiendo a la dimensión temporal, el ciclo de vida se divide en ciclos entendiendo estos como vueltas alrededor de una espiral, y estas, a su vez, como los períodos de tiempo en los que se trabaja sobre una versión completa del sistema. Cada ciclo se compone de cuatro fases que se realizan secuencialmente:

- **Fase de comienzo (*inception*):** el **objetivo** de esta fase es **estudiar la viabilidad del sistema**. Para ello, se establece el **objetivo del sistema** y se delimita su alcance, definiendo, además, las estimaciones de recursos y un plan de tiempos general en el que se establecen los **hitos principales**, las **previsiones financieras**, los **riesgos del proyecto** y los **criterios para su éxito**. Al finalizar esta fase, se decide si se continua o no con el proyecto.
- **Fase de elaboración (*elaboration*):** los **objetivos** de esta fase son **analizar el dominio del problema**, establecer una base de la arquitectura software, desarrollar el plan del proyecto y eliminar los **riesgos más importantes**. En esta fase **se debe disponer de un prototipo validado de la arquitectura software** a partir del cual se puede desarrollar el sistema.

- **Fase de construcción (construction):** durante esta fase se desarrolla el sistema de forma iterativa e incremental hasta que esté preparado para su puesta en funcionamiento. Para cada iteración, se seleccionan algunos casos de uso (requisitos), se refina su análisis y diseño y se procede a su implementación y pruebas. Se realizan tantas iteraciones como sean necesarias hasta que termine la implementación del producto. Como resultado de esta fase, se obtiene el sistema integrado en las plataformas adecuadas, los manuales de usuario y una descripción de la versión actual.
- **Fase de transición (transition):** el propósito de esta fase es poner en funcionamiento el sistema y ponerlo a disposición de los usuarios. En esta fase, suelen surgir nuevas cuestiones que requieren un desarrollo adicional para refinar y ajustar el sistema, corregir problemas o finalizar aspectos que puedan haber sido aplazados. Esta fase comienza con una versión beta del sistema, que se reemplaza finalmente con el sistema en producción.

Al término de cada una de estas fases, se define un hito principal, esto es, un punto en el tiempo en el que se deben haber conseguido unos objetivos determinados. Cuando llega este momento, la dirección del proyecto tiene que decidir si el trabajo puede continuar con la siguiente fase.

Cada una de las fases de RUP puede descomponerse en iteraciones. Una iteración es un periodo de tiempo en el que se realiza un conjunto completo de actividades de desarrollo. El sistema, por lo tanto, se va desarrollando incrementalmente de iteración en iteración. Una nueva iteración produce una nueva versión que proporciona al software mayor funcionalidad y es más refinada.

Atendiendo a la dimensión estática, el proceso RUP describe los perfiles o papeles de trabajo (quién) que realizan productos intermedios (qué) como resultado de realizar un conjunto de actividades (cómo) por medio de un flujo de trabajo predefinido (cuándo). De esta forma, el RUP establece cuatro elementos de modelado principales:

- **Perfiles o papeles de trabajo (workers):** mediante el papel o rol de trabajo se define el comportamiento y la responsabilidad de una persona o un grupo de personas que trabajan como una unidad y en equipo. Un individuo puede tener diversos roles y un rol lo pueden llevar a término varios individuos. Ejemplos de perfiles son la jefa o el jefe de un proyecto, las y los diseñadores de pruebas, los programadores y programadoras, etcétera.
- **Actividad (activity):** es una unidad de trabajo que lleva a cabo un individuo con un rol determinado. Por ejemplo, para el papel del jefe o jefa de proyecto, una actividad podría ser planificar una iteración. Otras actividades podrían ser: llevar a cabo pruebas, codificar una clase, detallar un caso de uso, etcétera.
- **Producto intermedio (artifact):** son elementos de información producidos, modificados o usados para el desarrollo del software. Son la entrada y la salida de las actividades y la responsabilidad de los perfiles o roles. A modo de ilustración, algunos productos intermedios podrían ser: un modelo de casos de uso, documentos, código fuente, ejecutables, etcétera.

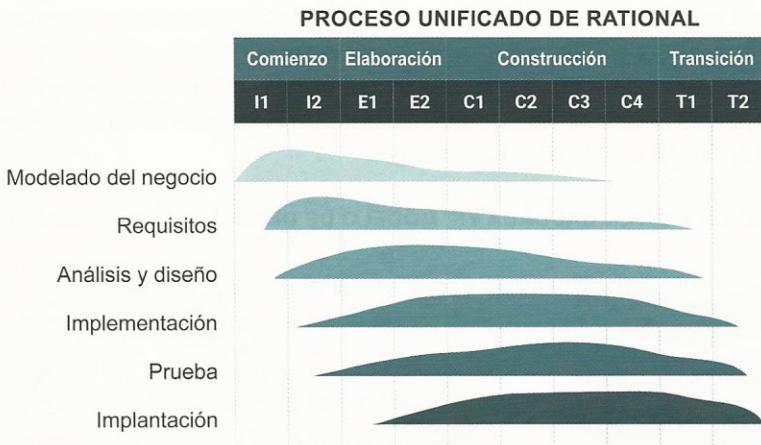
■ **Flujos de trabajo (workflows)**: son secuencias de actividades para producir productos intermedios. RUP define nueve flujos de trabajo agrupados en dos clases principales:

– **FLUJOS DE INGENIERÍA (actividades secuenciales):**

1. **Modelado del negocio (business modelling)**: el objetivo es entender el conjunto de procesos de negocio que aparecen dentro de la empresa como un paso previo a la recogida de requisitos del sistema que se debe desarrollar. En esta fase, se obtiene un modelo de casos de uso del negocio, que describe los procesos de negocio en términos de casos de uso y actores, y un modelo de objetos del negocio, que describe cómo cada caso de uso del negocio se lleva a cabo por el conjunto de trabajadores del negocio.
2. **Requisitos (requirements)**: el propósito es establecer los requisitos funcionales (qué debe hacer el sistema) mediante casos de uso y los requisitos no funcionales (rendimiento, restricciones, facilidad de mantenimiento, fiabilidad, etc.).
3. **Análisis y diseño (analysis & design)**: en esta fase se analizan los requisitos capturados anteriormente con el objeto de tener una comprensión y descripción más precisa de los mismos, se refinan y se estructuran. El diseño produce un modelo físico del sistema que se debe construir (modelo de diseño) y que no es genérico (específico para una implementación). Este debe ser más detallado que el de análisis y debe ser mantenido durante todo el ciclo de vida del software.
4. **Implementación (implementation)**: consiste en programar o implementar el sistema en términos de componentes, es decir, ficheros de código fuente, ejecutables, etcétera.
5. **Pruebas (test)**: incluye crear casos de prueba (especificando qué probar y cómo realizar la prueba), realizar las pruebas y evaluar sus resultados.
6. **Implantación (deployment)**: el objetivo es asegurarse de que el producto está preparado para ser suministrado al cliente, ajustar el producto de software a las necesidades del usuario y organizar su entrega y la recepción por parte del usuario.

– **FLUJOS DE APOYO (actividades de gestión paralelas):**

7. **Gestión de la configuración**: control de cambios sobre los productos intermedios.
8. **Gestión de proyecto**: planificación del proyecto, gestión de los riesgos y monitorización del progreso del proyecto.
9. **Entorno**: cubre la infraestructura necesaria para desarrollar un sistema.



**Figura 1.16.** Ejemplo de evolución de un proyecto de desarrollo empleando la metodología RUP. Se puede observar cómo en las diferentes fases (comienzo, elaboración, construcción y transición) se van realizando tareas correspondientes a los distintos flujos de ingeniería (modelado del negocio, requisitos, análisis y diseño, implementación, pruebas e implantación), así como el volumen de trabajo que supone cada uno de estos flujos de ingeniería en cada una de las iteraciones.

## ■■■ 1.9.2. Modelos de desarrollo ágil

En 2001, un grupo de desarrolladores encabezados por Kent Beck, grupo conocido como Alianza Ágil, firmaron el *Manifiesto por el desarrollo ágil de software*, en el que se establece lo siguiente (véase <https://agilemanifesto.org/>):

Estamos descubriendo formas mejores de desarrollar software, haciéndolo y ayudando a otros para que lo hagan. Este trabajo nos ha hecho valorar:

- Los individuos y sus interacciones, más que los procesos y las herramientas.
- El software que funciona, más que la documentación exhaustiva.
- La colaboración con el cliente, y no tanto la negociación del contrato.
- Responder al cambio, mejor que apegarse a un plan.

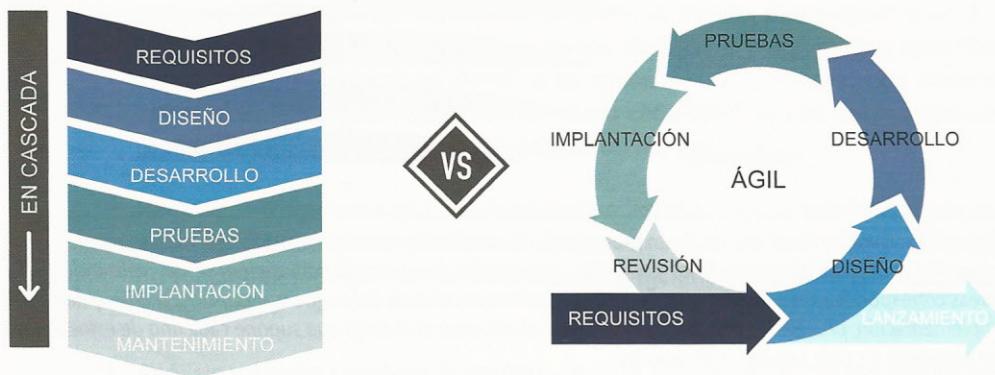
Es decir, si bien son valiosos los conceptos que aparecen en segundo lugar, valoramos más los que aparecen en primer lugar.

En el desarrollo ágil, se pone el **énfasis en la entrega** al cliente, cada poco tiempo, de software que funcione, más que en el rigor en la ingeniería del software y en los productos intermedios. Uno de los **objetivos** es la entrega rápida de software incremental y para ello se prefiere formar equipos pequeños y bien motivados.

Una de las **principales ventajas** del desarrollo ágil es su mayor facilidad para incorporar los cambios a lo largo del desarrollo y el menor coste que estos cambios conllevan. Como sabemos, cuanto más tarde en el proceso de desarrollo se detecta la necesidad de realizar un cambio, más costoso es incorporarlo. También resultará más costoso si

este cambio afecta a etapas iniciales (análisis o diseño). Gracias a la entrega incremental propia del desarrollo ágil y a otras prácticas propias de esta metodología, como las pruebas unitarias continuas y la programación por parejas, se consigue reducir el coste de los cambios. Esto también es posible gracias a las entregas cada poco tiempo y a la retroalimentación del cliente en cada entrega.

#### CICLO DE VIDA EN CASCADA VS. MODELO DE DESARROLLO INCREMENTAL



**Figura 1.17.** La agilidad, propia del modelo de desarrollo incremental, frente al modelo en cascada. En este modelo, se llevan a cabo las mismas tareas que en el modelo en cascada, pero se aplican sobre entregas pequeñas o incrementos de software.

La Alianza Ágil ha establecido doce principios de agilidad (véase <https://agilemanifesto.org/>), que son los siguientes:

1. La máxima prioridad es satisfacer al cliente a través de la entrega rápida y continua de software valioso.
2. Los requisitos cambiantes son bienvenidos, aun en etapas avanzadas del desarrollo, pues, cuando hay cambios, los procesos ágiles son beneficiosos para la ventaja competitiva del cliente.
3. Se realizan entregas frecuentes y lo más pronto que se pueda de software que funcione, preferiblemente entre dos semanas y un par de meses.
4. Las personas responsables del negocio y las encargadas del desarrollo del software trabajan conjuntamente, a diario y durante todo el proyecto.
5. Los proyectos son desarrollados por individuos motivados. Debe proporcionarse a estos el entorno y el apoyo adecuados, y confiar en que harán el trabajo.
6. El método más eficiente y eficaz para comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. La principal medida para progresar es un software que funcione.
8. Los procesos ágiles impulsan el desarrollo sostenible. Los promotores, las personas que desarrollan el software y las usuarias de este deben ser capaces de mantener un ritmo constante de forma indefinida.

9. La atención continua a la excelencia técnica y al buen diseño mejoran la agilidad.
10. Es esencial la simplicidad: el arte de maximizar la cantidad de trabajo no realizado.
11. Las mejores arquitecturas, requisitos y diseños surgen de los equipos con organización propia.
12. El equipo reflexiona con regularidad sobre cómo ser más eficaz, para después perfeccionar y ajustar su comportamiento en consecuencia.

No es necesario que todas las metodologías ágiles apliquen estos doce principios con la misma intensidad, pero estos principios sí que definen la filosofía del desarrollo ágil.



**Figura 1.18.** Cada uno de los incrementos de software se obtiene mediante una iteración, que en los modelos ágiles suele recibir el nombre de sprint. En este caso, se desarrolla el software en tres sprints.

Seguidamente, se presentan dos modelos concretos de desarrollo ágil:

### Programación extrema (XP)

La programación extrema (XP) fue formulada por Kent Beck y es el más destacado de los métodos de desarrollo ágil.

Los cinco valores originales de la programación extrema son la simplicidad, la comunicación, la retroalimentación, la valentía y el respeto. A continuación, se describen brevemente cada uno de ellos:

- **Simplicidad:** se debe simplificar el diseño para agilizar el desarrollo y facilitar el mantenimiento. Solo se diseña de forma inmediata lo necesario. Si se tuviera que mejorar el diseño, siempre se puede rediseñar o realizar una refactorización con posterioridad.
- **Comunicación:** debe establecerse una comunicación cercana pero informal con el cliente; debe existir una retroalimentación continua y se deben evitar los documentos voluminosos como medio de comunicación.
- **Retroalimentación:** al realizar incrementos de software en cortos intervalos de tiempo, los cuales son mostrados al cliente una vez terminados, se obtiene una retroalimentación continua y frecuente, lo que evita tener que rehacer partes importantes del trabajo. Las pruebas son una fuente esencial de la retroalimentación.

- **Valentía:** cumplir con las prácticas de la programación extrema requiere valentía. Por ejemplo, no se debe diseñar pensando en requisitos futuros, aunque llevar esto a la práctica es difícil, pues la mayoría de los equipos de desarrollo de software suelen recibir presión para que tengan en cuenta también los requerimientos futuros.
- **Respeto:** el cumplimiento de los cuatro principios anteriores conlleva respeto entre los miembros del equipo de desarrollo y entre estos y las demás personas que participan en el proyecto. Así, las personas responsables de la programación no pueden realizar cambios que hagan que las pruebas existentes fallen o que retrasen el trabajo de sus compañeras y compañeros.

La programación extrema usa el enfoque orientado a objetos como paradigma de desarrollo preferido.

Por otro lado, en la estructuración del proceso de desarrollo de software, se llevan a cabo las siguientes **cuatro actividades estructurales**: planificación, diseño, codificación y pruebas:

- **Planificación:** tras escuchar al cliente, se crean las llamadas **historias de usuario**, similares a los casos de uso, que son escritas por el cliente; se colocan entonces en una tarjeta y el cliente les asigna una prioridad. Posteriormente, el equipo XP asigna a cada historia un coste (semanas de desarrollo). El tiempo máximo de una historia es de tres semanas; en caso de que sea más larga, se le pide al cliente que la descomponga en historias más pequeñas y, de nuevo, se le asigna una prioridad y un coste. En cualquier momento, se pueden escribir nuevas historias. El cliente y las personas que se encargan del desarrollo deciden conjuntamente qué historias incluir en el siguiente incremento de software, así como su fecha de entrega y demás aspectos relevantes.
- **Diseño:** la XP contribuye al diseño sencillo. Cuando hay dificultades en el diseño, se suele crear un prototipo que permita disminuir el riesgo. La XP fomenta también el rediseño o refactorización, que consiste en reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad, pero sin modificar su comportamiento. Se deben realizar pruebas para garantizar que al refactorizar no se hayan introducido errores.
- **Codificación:** antes de la codificación, se diseñan pruebas unitarias para cada una de las historias. Una vez creado el código, se aplica la prueba unitaria creada para proporcionar retroalimentación a las personas que lo han desarrollado. Un concepto clave de la codificación es la **programación por parejas**, en la que cada persona juega un rol distinto: por ejemplo, uno se centra en los detalles del código y el otro se encarga de asegurarse de que se siguen las pautas de codificación y de que el código superará con éxito la prueba unitaria. A medida que las parejas de programadores terminan su trabajo, el código que desarrollan se integra con el trabajo de los demás.
- **Pruebas:** además de las pruebas unitarias que se hacen una vez creado el código, también se realizan continuamente pruebas de integración y validación, lo que

revela al equipo XP los avances obtenidos de forma continua y permite lanzar señales de alerta en caso de que haya problemas importantes. Finalmente, se llevan a cabo las pruebas de aceptación, que son especificadas por el cliente.

## Scrum

Scrum es un modelo de desarrollo ágil concebido por Jeff Sutherland y su equipo de desarrollo en la década de los 90 y que luego fue desarrollado por Schwaber y Beedle. Este modelo de desarrollo ágil tiene como objetivo la entrega de valor (productos) al cliente en cortos períodos de tiempo y se basa en tres pilares: transparencia, inspección y adaptación:

- **Transparencia:** todas las personas involucradas en el proyecto conocen en todo momento qué ocurre y cómo, lo que permite que haya una visión global del proyecto.
- **Inspección:** todos los miembros del equipo inspeccionan de manera autoorganizada el progreso del proyecto para detectar posibles problemas.
- **Adaptación:** cuando es necesario realizar algún cambio en el proyecto, el equipo se adapta para conseguir el objetivo.

En esta metodología, se consideran las siguientes actividades estructurales: *requerimientos, análisis, diseño, evolución y entrega*. Cada actividad estructural se divide en una serie de sprints. Un sprint abarca el periodo de tiempo que se emplea para realizar el trabajo necesario para entregar valor al cliente. La duración máxima de un sprint es de un mes, aunque es aconsejable que sea inferior (unas dos semanas). La duración se determina en función del nivel de comunicación que el cliente desea tener con el equipo de desarrollo.

Antes de iniciarse un sprint, tiene lugar una reunión llamada *sprint planning*, en la que todo el equipo establece qué tareas se van a realizar en el sprint, cuál es su objetivo y cómo se van a llevar a cabo las tareas. Al final del sprint, se hace una entrega de valor al cliente. Se organizan reuniones diarias dentro de cada sprint con una duración máxima de 15 minutos. En las reuniones, se responde a las siguientes preguntas: ¿qué hicimos ayer?, ¿qué vamos a hacer hoy? Y ¿tenemos algún problema que hay que solucionar?

Una vez ha concluido el sprint, se entrega y presenta el producto al cliente (*sprint review*). El equipo de desarrollo muestra el funcionamiento del producto al cliente, quien tendrá que validararlo. Además, durante esta reunión, el cliente proporciona retroalimentación útil para nuevas tareas.

Tras el *sprint review*, se efectúa una *retrospectiva del sprint*, en la que se hace una evaluación del trabajo realizado durante el sprint, proponiendo mejoras para el siguiente sprint. Finalizada esta reunión, se comienza el siguiente sprint, que incluirá las cuatro tareas que se han indicado: planificación, reuniones diarias, *sprint review* y retrospectiva.

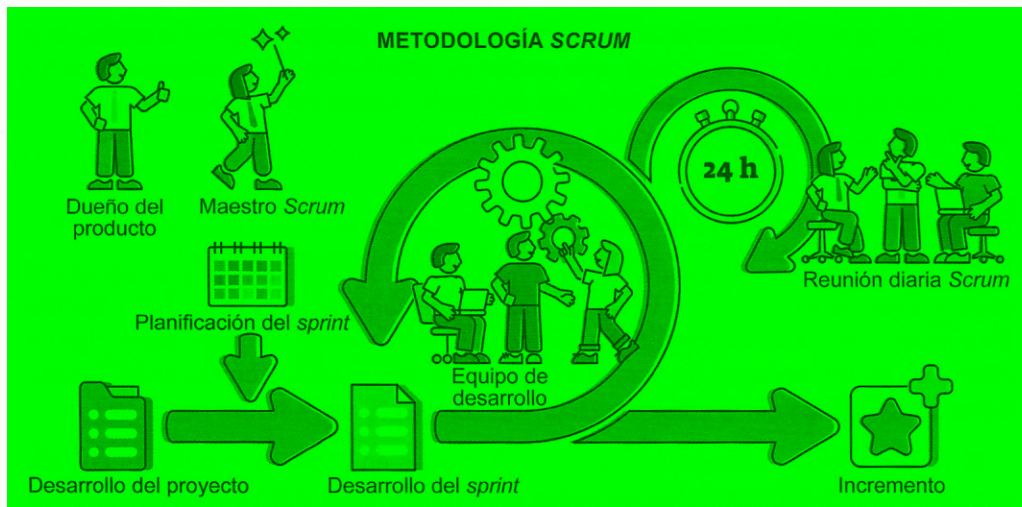


Figura 1.19. En la metodología Scrum todo el trabajo que hay que realizar para el desarrollo del proyecto (product backlog) se divide en una serie de sprints. A partir de la planificación del sprint, se crea una lista de tareas (sprint backlog) que irá realizando el equipo asignado (Scrum team). Cada día de trabajo, se lleva a cabo una reunión para analizar la marcha del sprint (Scrum meeting) y, al finalizar el sprint, se realiza una presentación al cliente (sprint review) del producto obtenido hasta el momento (incremento, en inglés 'product increment').

## Actividad resuelta 1.2

### Los modelos de desarrollo ágil

Indica las características más relevantes que distinguen a los modelos de desarrollo ágil frente al resto de metodologías.

#### Solución

Las características diferenciales de mayor relevancia de los modelos de desarrollo ágil frente a otras metodologías son:

- Se sigue un proceso de desarrollo menos burocrático, en el que se da más relevancia al software que funcione y menos importancia a la documentación.
- Se considera fundamental la colaboración con el cliente y su implicación constante a lo largo de todo el proceso de desarrollo.
- Se hacen cada poco tiempo entregas de software que funciona al cliente, con el fin de que este lo pueda evaluar y proporcionar así una retroalimentación gracias a la cual se puedan solventar errores o problemas lo antes posible.
- Gracias a las dos características indicadas anteriormente, es más sencilla la incorporación de cambios al software.
- El equipo reflexiona frecuentemente sobre el proceso de desarrollo con vistas a mejorar su funcionamiento.