# Introducción a la Ingeniería del Software

Jesús Sánchez Cuadrado
Última actualización: 26/01/2026

En este tema se introducen los conceptos básicos de la Ingeniería del Software, incluyendo las características del software, la definición de Ingeniería del Software y sus principios y técnicas básicas. También se resumen algunas tendencias actuales.

## Índice



## 1. Introducción

En este primer tema se van a tratar algunas cuestiones básicas sobre las que se profundizará en el resto del curso.

En primer lugar, vamos a ver cuáles son las características del software que hacen que esta disciplina sea compleja y que requiera soluciones especiales.

En segundo lugar, vamos a ver qué es la Ingeniería del Software. Como veremos, la Ingeniería del Software nos va a dar un marco para poder abordar la complejidad del software (y tener éxito en nuestros proyectos).

Por último, se repasarán algunas tendencias actuales en el desarrollo de software.

## 2. El software y sus características

La primera cuestión que uno debería plantearse cuando empieza a estudiar Ingeniería del Software es: ¿qué tiene el software que lo hace tan especial? ¿Por qué no podemos usar las mismas técnicas que se usan en otras ingenierías (civil, mecánica, eléctrica, etc.)?

Un elemento clave que diferencia al software de otros productos de ingeniería es su naturaleza lógica y abstracta.

> El software es el conjunto de instrucciones y datos que indican a una computadora cómo realizar ciertas tareas.

Esta definición es algo reduccionista porque en la práctica el software no son solo los programas, sino también todo lo que hay alrededor de un sistema software para su operación y mantenimiento. Esto incluye, el código fuente, las herramientas de desarrollo, los entornos de ejecución, la documentación, las pruebas, la formación y soporte, etc.

### 2.1. Características del software

El software tiene una serie de características que lo hacen diferente, y posiblemente más complejo, que otros productos de ingeniería. En particular, y comparando con el hardware:

- El software no se fabrica, sino que se desarrolla. En cierto modo se puede considerar *"un craft"* o artesanía. Al contrario que el hardware,

el software no se construye en una línea de producción. Cada sistema software es único y se desarrolla específicamente para un propósito o cliente concreto.

- Es un elemento lógico, no físico. Es decir, cualquier cosa que imaginemos se puede hacer realidad en software[1].
- El software es *"maleable"*. Se puede cambiar fácilmente, y eso provoca cierto "incentivo" para hacerlo[2]. De alguna forma, el software se cambia porque se puede.

[1] Asumiendo que dispongamos del tiempo suficiente, infinito en el peor de los casos.

[2] Sin embargo, a nadie se le ocurriría cambiar un rascacielos una vez construido.

Por otra parte, el software no se corrompe porque no es un elemento físico. Sin embargo, se deteriora por diversas razones:

- Cambios en el software, debido a cambios en los requisitos.
- Deuda técnica, es decir, la dificultad añadida para introducir cambios o adaptaciones debido a que originalmente se tomaron malas decisiones.
- Cambios en los entornos de ejecución. La realidad física rara vez cambia, pero la tecnológica lo hace muy rápido.

Es interesante observar qué sucede con los **fallos del software**. Uno esperaría que durante el desarrollo se fueran corrigiendo todos los fallos hasta que ya no hubiera y el software siempre operase bien. Sin embargo, en la práctica lo que sucede es que siempre se introducen cambios por una u otra razón, y nos arriesgamos a deteriorar el software cada vez más.

La Figura 1 muestra un ejemplo típico de la evolución de los fallos en un sistema software a lo largo del tiempo.

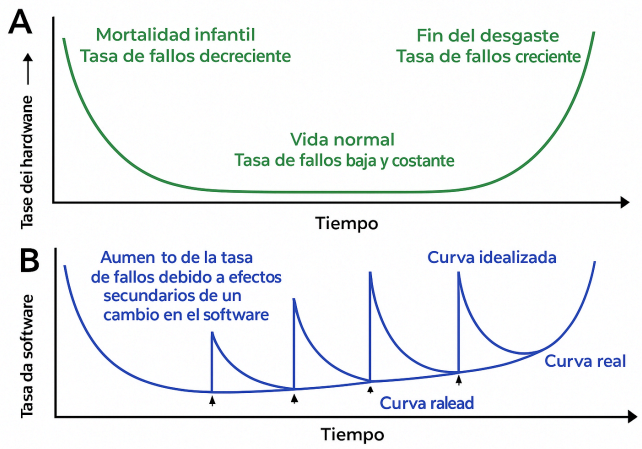


Figura 1: Evolución de los fallos en el software a lo largo del tiempo.

## 2.2. Problemas en el desarrollo de software

A la hora de de desarrollar software existen una serie de problemas recurrentes [1].

- ¿Por qué lleva tanto tiempo terminar los programas?
- ¿Por qué es tan elevado el coste de desarrollo?
- ¿Por qué no es posible entregar todos los errores antes de entregar el software?

- ¿Por qué resulta tan difícil hacer cambios en el software una vez que está terminado?
- ¿Por qué resulta dificil constatar el progreso de un proyecto de software?

Hay algunas causas que explican estos problemas. Unas de ellas son **esenciales** mientras que otras son **accidentales** [2].

Las causas esenciales son aquellas que derivan de la propia naturaleza del software. Algunas de estas causas tienen que ver con las características propias del software que hemos visto antes:

- La naturaleza lógica del software.
- El software se puede cambiar fácilmente, y eso provoca cierto "incentivo" para hacerlo.

Sin embargo, también existen causas accidentales, que derivan de las técnicas y herramientas que usamos para desarrollar software. Algunas de estas causas son:

- Problemas derivados de la comunicación con los clientes.
- Poco esfuerzo en análisis y diseño.
- Problemas de gestión de proyectos.
- Ausencia de guías claras. Cada empresa o equipo usa sus propias técnicas y herramientas, lo que dificulta la estandarización y la mejora continua.
- Ausencia de estándares, aunque se han establecido algunos.
- Ausencia de base formal.

Se han propuesto algunas **soluciones** pero solo aquellas que aborden la complejidad esencial del software pueden tener éxito a largo plazo.

## 3. Ingeniería del Software

Hasta ahora se han discutido las características del software y cómo estás características hacen que el desarrollo de software sea complejo y propenso a errores. Ahora vamos a ver qué es la Ingeniería del Software y cómo nos puede ayudar a abordar esta problemática.

### 3.1. Definición de Ingeniería del Software

La **Ingeniería del Software** es la aplicación sistemática de un proceso de diseño, desarrollo, prueba y mantenimiento del software. Es un enfoque sistemático y disciplinado para el desarrollo de software que tiene como objetivo crear software de alta calidad, confiable y fácil de mantener.

Hay muchas más definiciones de Ingeniería del Software, pero todas coinciden en:

- Aplicar principios ingenieriles.
- Usar un enfoque sistemático y disciplinado.
- Obtener software de calidad.

## 3.2. Principios de la Ingeniería del Software

Existen una serie de principios y técnicas fundamentales en la Ingeniería del Software que nos ayudan a manejar la complejidad y mejorar la calidad del software que construimos. Algunos de estos principios son:

- Abstracción
- Modularidad
- Encapsulación
- Alta cohesión y bajo acoplamiento

Un aspecto importante es que estos principios muchas veces no son independientes entre sí, si no que son fuerzas en tensión (trade-offs). Por ejemplo, la abstracción puede ayudar a mejorar la modularidad, pero a costa de perder algo de rendimiento o incluso empeorar la legibilidad[3].

[3] Ver ejemplo en https://www.hillelwayne.com/post/what-comments/

### 3.2.1. Abstracción

La **abstracción** hace referencia al proceso de ocultar los detalles irrelevantes para centrarse en los aspectos esenciales y relevantes.

Existing muchos tipos de abstracciones en el software, como por ejemplo:

- Funciones y procedimientos, que abstraen un conjunto de instrucciones en una unidad reutilizable.
- Construcciones como bucles que abstraen la forma en que se itera.
- Clases y objetos, que abstraen entidades del mundo real en estructuras de datos con comportamiento

Decimos que algo es **una abstracción** cuando utilizamos conceptos de un lenguaje programación para representar algo de manera simplificada. En programación orientada a objetos, la abstracción suele materializarse mediante interfaces y clases. Por ejemplo, una interfaz `List` abstrae el concepto de lista, sin importar cómo se implemente internamente (array, lista enlazada, etc.).

El objetivo de la abstracción es reducir la complejidad cognitiva, permitiendo razonar sobre el sistema sin tener que entender todos los detalles de implementación.

**Cuestiones a tener en cuenta**:

- Abstracción vs. rendimiento: A veces, las abstracciones pueden introducir una sobrecarga de rendimiento debido a la capa adicional de indirection.
- Abstracción vs. legibilidad: Demasiada abstracción puede dificultar la comprensión (hay que saltar entre clases o métodos para entender qué hace algo).

Joel Spolsky acuño el término ***"leaky abstractions"*** para referirse a las abstracciones que no ocultan completamente los detalles subyacentes: ***All non-trivial abstractions, to some degree, are leaky.*** https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

Un ejemplo muy claro es el bucle **for**: el orden de iteración tiene un impacto muy grande en el rendimiento porque afecta al uso de la memoria caché.

### 3.2.2. Modularidad

La **modularidad** consiste en dividir un sistema en partes o módulos independientes, cada uno con una responsabilidad bien definida.

Un módulo es una unidad que encapsula una funcionalidad específica y puede interactuar con otros módulos a través de interfaces bien definidas.

Los beneficios esperados de la programación modular son [3]:

- Es posible dividir el trabajo porque los módulos pueden desarrollarse de manera independiente.
- Es posible hacer cambios en un módulo sin afectar a otros.
- Es posible entender el sistema estudiando un módulo a la vez.

### 3.2.3. Encapsulación

La **encapsulación** consiste en ocultar el estado interno y los detalles de implementación de un módulo, exponiendo únicamente una interfaz controlada para interactuar con él.

El objetivo es:

- Proteger los invariantes del sistema
- Reducir el acoplamiento entre módulos
- Permitir cambios internos sin afectar al código cliente

```
public class CuentaBancaria {
    public double saldo;
}

var cuenta = new CuentaBancaria();
cuenta.saldo = -1000; // ¡Error!
```

Modularidad y encapsulación están relacionadas pero no son lo mismo. La modularidad es acerca de cómo dividir el sistema, mientras que la encapsulación es acerca de cómo proteger los detalles internos de cada parte. En ambos casos se diseña la interfaz de los módulos. De alguna forma, la modularidad trabaja a nivel macro y la encapsulación a nivel micro.

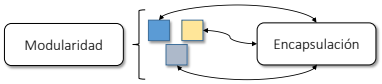

Figura 2: Modularidad vs. Encapsulación.

⚠️ **Importante**

Cuando hablamos de encapsulación en OO no nos refererimos a “poner getters y setters” sino a diseñar buenas interfaces que reflejen las operaciones del dominio y protegan los invariantes de la clase.

### 3.2.4. Cohesion y acoplamiento

La cohesión y el acoplamiento son dos principios fundamentales de la ingeniería del software que van de la mano. El objetivo es lograr **módulos con alta cohesión y bajo acoplamiento**.

La **cohesión** se refiere al grado en que los elementos dentro de un módulo están relacionados y trabajan juntos para cumplir una única responsabilidad o propósito.

Si un módulo tiene una única responsabilidad bien definida, se dice que tiene alta cohesión.

Si un módulo realiza múltiples tareas no relacionadas, se dice que tiene baja cohesión.

Un ejemplo de módulo con baja cohesión son las típicas clases “manager” donde se agrupan métodos sin relación entre sí.

```
public class GestorDeUsuarios {

    public void crearUsuario() { }
    public void eliminarUsuario() { }
```

```
    public void enviarMensajeAUsuario() { }
    public void generarInforme() { }
    public void conectarBD() { }
}
```

El **acoplamiento** se refiere al grado de dependencia entre diferentes módulos o componentes de un sistema.

Si un módulo interactua con otros módulos a través de interfaces claras y estables y sin depender de los detalles de los otros módulos, se dice que tiene bajo acoplamiento.

Si un módulo depende fuertemente de los detalles internos de otros módulos, se dice que tiene alto acoplamiento.

Un bajo acoplamiento favorece:

- Cambios localizados
- Sustitución de módulos
- Pruebas independientes

Por ejemplo, en el siguiente código el `ServicioPedidos` tiene un alto acoplamiento con `RepositorioMySQL` porque depende directamente de su implementación concreta. Para probar este código es necesario configurar una base de datos MySQL como en el sistema real.

```
public class ServicioPedidos {

    private RepositorioMySQL repo = new RepositorioMySQL();

    public void crearPedido() {
        // lógica para crear pedido
        repo.save();
    }
}
```

## 3.3. Técnicas y buenas prácticas en la Ingeniería del Software

Algunas de estas técnicas y buenas prácticas como:

- Reutilización
- Patrones de diseño
- Arquitectura del software
- Pruebas de software
- Análisis estático

### 3.3.1. Reutilización

La reutilización de software es el proceso de crear sistemas de software a partir de (piezas de) software existente en lugar de construir el sistemas desde cero.

La reutilización de software tiene dos caras: la creación de software *con reuso* (utilizando componentes existentes) y la creación de componentes que puedan ser reutilizados en el futuro (*para reuso*) [4]

Hoy en día ambas aproximaciones se retroalimentan, existiendo grandes ecosistemas de librerías reutilizables (por ejemplo, npm para JavaScript, PyPI para Python, Maven para Java, etc.). La mayoría de los proyectos de software hacen uso intensivo de estas librerías y componentes externos.

La construcción de software reutilizable se basa en los principios de abstracción, modularidad y encapsulación que hemos visto antes.

### 3.3.2. Patrones de diseño

Un **patrón de diseño** es una solución reutilizable para resolver cierto tipo de problema en el desarrollo de software.

Un patrón de diseño no es algo que sea pueda aplicar directamente, sino una especie de "plantilla" o "guía" que describe cómo resolver un problema de diseño en términos generales.

Los patrones más conocidos son los descritos por Gamma et al. en el libro "Design Patterns: Elements of Reusable Object-Oriented Software" [5]: estrategia, observador, factory, decorador, adaptador, fachada, builder, interpreter, visitor, etc.

Algunos de estos patrones no hacen falta como tal cuando los lenguajes añaden características que abordan la misma problemática o quizás la implementación del patrón cambia según estas características. Por ejemplo, a partir de Java 8 es posible implementar (algunas) estrategias utilizando *lambdas*. En Kotlin el patrón singleton se puede implementar fácilmente usando un `object`. En Kotlin o Ruby, el patrón Builder se puede implementar utilizando DSLs internos. También, las factorías/singleton pueden ser sustituidas por inyección de dependencias proporcionadas por frameworks como Spring o Guice.

**❓ Idiomas o patrones de diseño**

¿Cuál es la diferencia en un *idiom* y un patrón de diseño?

Un *idiom* es una solución de bajo nivel, a veces específica de un lenguaje de programación concreto, mientras que un patrón de diseño es una solución de más alto nivel, aplicable a múltiples lenguajes y contextos.

Ejemplo:

```
*dst++ = *src++
```

Ejemplo:

```
String sep = "";
for (String item: lista) {
    println(separator + item);
    sep = ", ";
}
```

### 3.3.3. Arquitectura del software

La arquitectura del software se refiere a la estructura general de un sistema de software, incluyendo los componentes principales, sus interacciones y las decisiones de diseño que guían su desarrollo y evolución.

Igual que existen patrones a nivel de diseño, también existen patrones a nivel arquitectónico. Algunos ejemplos son:

- Arquitectura en capas
- Arquitectura basada en eventos
- Arquitectura hexagonal
- Arquitectura de microservicios

### 3.3.4. Pruebas de software

Si desarrollamos el software simplemente escribiendo todo el código "de una" es casi seguro que contendrá errores. Podríamos probarlo manualmente pero es costoso y no es repetible (¿podemos asegurar que siempre probamos lo mismo?).

Por eso, una práctica fundamental en la Ingeniería del Software es la realización de pruebas automáticas. El objetivo de las pruebas de software es asegurar que el software funciona correctamente en diferentes escenarios.

Las pruebas y el diseño están muy relacionadas. Un buen diseño facilita la realización de pruebas, y las pruebas ayudan a mejorar el diseño.

En relación con las pruebas de software, una metodología de desarrollo muy popular es el denominado TDD (Test-Driven Development). Esencialmente consiste en escribir las pruebas antes que el código, de manera que el diseño del software se ve influenciado por las pruebas que se van a realizar.

### 3.3.5. Análisis estático

El análisis estático es una técnica que consiste en examinar el código fuente sin ejecutarlo para detectar posibles errores, vulnerabilidades o incumplimientos de estándares de codificación.

### 3.3.6. Integración continua (CI)

La integración continúa (CI) es una práctica de desarrollo de software que consiste en que los miembros de un equipo integran sus cambios en un proyecto con frecuencia (ej., una vez al día al menos). La clave es que cada integración se verifica automáticamente utilizando pruebas, análisis estático, etc. [4]

[4] https://martinfowler.com/articles/continuousIntegration.html

### 3.3.7. Frameworks

> Un framework es una plataforma que proporciona un conjunto de herramientas, librerías y convenciones para facilitar el desarrollo de aplicaciones de software.

Los frameworks se pueden clasificar en:

- Frameworks horizontales cuando son generalistas y pueden aplicarse a muchos dominios o problemas. En general, un framework de este tipo aborda un cierto aspecto del ***stack*** tecnológico. Por ejemplo, para desarrollo web Spring o Django, para aplicaciones móviles Android, etc.
- Frameworks verticales cuando están especializados en un dominio o industria concreta. En este caso abordan varias partes de la tecnología para ofrecer una solución completa pero personalizable. Por ejemplo, para gestión empresarial SAP, para comercio electrónico Magento, etc.

## 4. Cómo se construye software hoy día

La forma en que se construye software ha evolucionado mucho en las últimas décadas y especialmente en los últimos años. Las técnicas y tecnologías modernas permiten construir sistemas anteriormente inimaginables en una fracción del tiempo que costaba antes, aunque los principios fundamentales para conseguir software de calidad siguen siendo similares. .

### 4.1. Modularidad y arquitecturas modernas

Las aplicaciones modernas suelen estructurarse de manera muy modular, muchas veces organizadas alrededor de servicios independientes. Estos servicios muchas veces están distribuidos (microservicios) y se comunican entre sí a través de APIs bien definidas.

💡 **Technology radar**

Un buen recurso para ver las tendencias actuales en el desarrollo de software es el ***Technology Radar*** de ThoughtWorks: https://www.thoughtworks.com/radar

## 4.2. Reutilización y ecosistema open source

La reutilización es un principio clásico que adquiere una nueva dimensión en el desarrollo actual. Hoy en día la reutilización se produce principalmente a través del uso intensivo de librerías y frameworks open source.

Una aplicación Java moderna depende habitualmente de decenas de componentes open source: frameworks de desarrollo, librerías de acceso a datos, herramientas de serialización, seguridad, pruebas, logging, etc. Todos estos componentes se integran en el proyecto utilizando gestores de dependencias como Maven o Gradle.

Por ejemplo, si vamos a desarrollar una aplicación móvil usaremos la API de Android, y al hacerlo estamos reutilizando 12 millones de líneas de código[5].

En este contexto, el trabajo del ingeniero de software se desplaza parcialmente desde la implementación desde cero hacia la selección, integración y mantenimiento de dependencias externas..

Este enfoque introduce nuevas preocupaciones de Ingeniería del Software, como la gestión de versiones, la compatibilidad entre componentes, la evaluación de la calidad del software reutilizado, las licencias y la seguridad de las dependencias.

La cuestión es que la mayoría del código que usamos no lo hemos escrito nosotros mismos, pero aún así tenemos que entender las implicaciones que tiene su uso en nuestro proyecto.

[5] El software libre ha sido un ***driver*** fundamental para la explosión del desarrollo de software

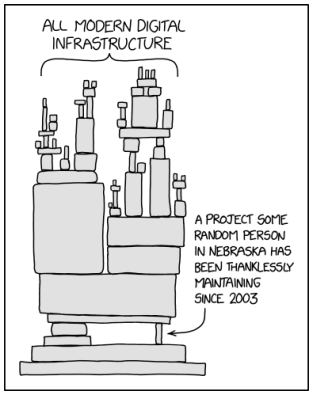


Figura 3: https://www.explainxkcd.com/wiki/index.php/2347:_Dependency.

## 4.3. Automatización, verificación y validación

La verificación y validación del software ya no son actividades que se realizan al final del desarrollo o puntualmente cuando un desarrollador lo considera oportuno.

Las pruebas automatizadas y el análisis estático, se ejecutan de manera sistemática mediante herramientas de integración continua que permiten verificar que el sistema "es correcto"[6] tras cada cambio.

Por otro lado, la validación se ve reforzada por la posibilidad de desplegar versiones funcionales del sistema con frecuencia (utilizando técnicas de despliegue continuo) y obtener feedback real de los usuarios. De este modo, se reduce el riesgo de construir un sistema técnicamente correcto pero que no satisface las necesidades reales.

[6] No es posible garantizar totalmente que un sistema es correcto con pruebas automatizadas pero cuanto mejor sean las pruebas más probable es que el software sea correcto.

## 4.4. Desarrollo para la nube

El desarrollo de software se ha adaptado a la era de la computación en la nube. Muchas aplicaciones modernas se diseñan y despliegan para ejecutarse en entornos de nube, aprovechando las ventajas que ofrecen en términos de escalabilidad, disponibilidad y flexibilidad.

En este contexto las arquitecturas que favorecen la distribución (ej., microservicios) y el uso de contenedores (ej., Docker) para el despliegue y la gestión de las dependencias se han convertido en prácticas comunes.

### 4.5. Desarrollo iterativo y gestión del cambio

El método de desarrollo más común hoy en día es el desarrollo iterativo e incremental, a menudo dentro del marco de metodologías ágiles.

Sin embargo, es conveniente no perder de vista las tecnicas de desarrollo en cascada tradicionales, por varias razones. Primero, siguen siendo necesarios en ciertos tipos de proyectos, especialmente aquellos con requisitos muy estables y bien definidos o en dominios críticos en que los proveedores necesitan comunicarse a través de contratos muy bien definidos. Segundo, con las técnicas actuales de IA es posible que se vea un resurgimiento de las "técnicas en cascada" donde el esfuerzo está en el diseño inicial.

### 4.6. Uso de la inteligencia artificial

En los últimos años se ha incorporado de forma creciente el uso de inteligencia artificial como apoyo al desarrollo de software, especialmente a través de modelos de lenguaje de gran tamaño (Large Language Models, LLMs).

La tendencia más actual es el uso de agentes basados en IA. Una agente es capaz de realizar acciones de manera autónoma (por ejemplo, modificar código, ejecutar pruebas y analizar resultados) dentro de un entorno de desarrollo controlado.

Desde el punto de vista de la Ingeniería del Software, estas herramientas no sustituyen los principios clásicos de diseño, verificación o mantenimiento, sino que actúan como amplificadores de la productividad del ingeniero. Al contrario, el uso de IA require una mayor atención a los principios básicos de la Ingeniería del Software, por varias razones:

- Las intrucciones (prompts) deben diseñarse cuidadosamente para guiar a la IA a producir resultados correctos y útiles (el viejo principio de "garbage in, garbage out").
- Es necesario revisar el código generado por la IA para asegurar su calidad y corrección[7]. Es importante tener en cuenta que los LLMs muchas veces generan código que parece correcto pero que en realidad contiene errores sutiles.

[7] https://www.germandz.com/blog/software-development/code-review-ai-era

## Bibliografía

[1] R. S. Pressman, ***Ingeniería del software un enfoque práctico***, 9a edición. McGrawHill, 2023.

[2] F. Brooks y H. Kugler, ***No silver bullet***. April, 1987.

[3] D. L. Parnas, «On the criteria to be used in decomposing systems into modules», ***Communications of the ACM***, vol. 15, n.º 12, pp. 1053-1058, 1972.

[4] C. W. Krueger, «Software reuse», ***ACM Computing Surveys (CSUR)***, vol. 24, n.º 2, pp. 131-183, 1992.

[5] E. Gamma, R. Helm, R. Johnson, y J. M. Vlissides, ***Design Patterns: Elements of Reusable Object-Oriented Software***. Addison-Wesley Professional, 1994.