# Temario Fundamentos de Ingeniería del Software

**Ismael Sallami Moreno**

ism350zsallami@correo.ugr.es

https://ismael-sallami.github.io/

https://elblogdeismael.github.io/

**Universidad de Granada**

**2025**

# Licencia

# Índice general

# Capítulo 1

# Introducción a la Ingeniería del Software

## 1.1. El producto software, propiedades y ciclo de vida

El software no se parece a los productos que fabrica el resto de las ingenierías. No se gasta con el uso, no está sujeto a tolerancias de material y su coste de replicación es prácticamente nulo: una copia vale lo mismo que un millón. Lo que cuesta es concebirlo y mantenerlo, y ahí es donde se concentra el esfuerzo de la disciplina.

De esa naturaleza salen dos consecuencias que condicionan todo lo demás. La primera es que no hay límites físicos que frenen la complejidad. Un puente no puede crecer sin que la estructura lo acuse; un sistema software sí, y por eso puede llegar a un tamaño que ninguna persona abarca. La segunda es que el software no se deteriora, pero sí se degrada: cada cambio que se le hace lo aleja del diseño con el que nació, y la facilidad para modificarlo acaba siendo la causa de que se vuelva difícil de modificar [1].

Conviene separar dos términos que se usan como sinónimos y no lo son. El **producto software** es lo que se entrega: los programas, los datos de configuración y la documentación que permite instalarlo, usarlo y mantenerlo. El **sistema software** incluye además el entorno en el que ese producto opera. Entregar solo los ejecutables no es entregar el producto.

### Propiedades de un producto software

La calidad de un producto software no se mide únicamente por que haga lo que se le pide. Las propiedades que suelen exigirse son:

- **Corrección y fiabilidad.** Que haga lo especificado y que siga haciéndolo en condiciones adversas, incluidas las que no se previeron.
- **Mantenibilidad.** Que pueda evolucionar con un coste razonable. Es la propiedad que más peso tiene a largo plazo, porque el grueso del gasto de un sistema se produce después de la primera entrega.
- **Eficiencia.** Que use de forma razonable el tiempo de cómputo, la memoria y el resto de recursos.

- **Usabilidad.** Que las personas a las que va dirigido puedan usarlo sin formación desproporcionada.

- **Seguridad y protección.** Que no cause daños y que resista un uso malintencionado.

Estas propiedades compiten entre sí. Un diseño muy eficiente suele ser menos legible y, por tanto, menos mantenible; una interfaz muy flexible complica la validación. Parte del trabajo de ingeniería consiste precisamente en decidir qué se sacrifica y dejar constancia de por qué.

#### El ciclo de vida

El ciclo de vida es el conjunto de etapas por las que pasa un producto desde que se identifica la necesidad hasta que se retira. Con distintos nombres, casi todos los modelos reconocen las mismas actividades: especificación, diseño e implementación, validación y evolución.

Lo que distingue a un modelo de otro no son las actividades, sino cómo se ordenan y con qué frecuencia se repiten. Merece la pena insistir en la última etapa: **la evolución no es un apéndice del ciclo de vida, es la mayor parte de él**. Un sistema en producción recibe cambios durante años, y el diseño con el que se construyó determina lo caros que resultan.

## 1.2. Concepto de Ingeniería del Software

El término nació en la conferencia de la OTAN de 1968, convocada porque los proyectos software se entregaban tarde, por encima del presupuesto y sin la calidad prometida. La expresión «crisis del software» describía esa situación, y la propuesta fue tratar el desarrollo como una ingeniería: con métodos, medidas y procesos, en lugar de como un oficio artesanal.

La Ingeniería del Software es, en esa línea, la disciplina que se ocupa de todos los aspectos de la producción de software, desde las primeras etapas de la especificación hasta el mantenimiento posterior a la entrega [1]. Dos matices de esa definición importan:

- **Todos los aspectos**, no solo la programación. La gestión del proyecto, la elección de herramientas y la organización del equipo forman parte de la disciplina.

- **Hasta el mantenimiento**, no hasta la entrega. Un método que funciona bien hasta el día del despliegue y deja el sistema intratable después no ha resuelto el problema.

Conviene distinguirla de dos disciplinas vecinas. La **informática** estudia los fundamentos teóricos del cómputo; la ingeniería del software se ocupa de construir sistemas útiles con esos fundamentos, aceptando restricciones de plazo, coste y personas. La **ingeniería de sistemas** es más amplia: abarca el hardware, los procesos y la organización, y el software es solo una de sus partes.

### Responsabilidad profesional

La disciplina lleva asociada una dimensión ética que no es decorativa. Quien construye software decide, en la práctica, sobre la privacidad de terceros, sobre la accesibilidad de un servicio y, en sistemas críticos, sobre la seguridad de las personas. Los códigos deontológicos de ACM e IEEE recogen los compromisos habituales: confidencialidad, competencia declarada con honestidad, respeto a la propiedad intelectual y no aceptar encargos que se sepan perjudiciales [1].

## 1.3. El proceso de desarrollo de software

Un proceso de desarrollo es un conjunto ordenado de actividades que produce un producto software. Los modelos que se estudian a continuación no son recetas excluyentes: en un proyecto real se combinan, y la elección depende de cuánto se conoce el problema al empezar y de cuánto cuesta equivocarse.

### Modelo en cascada

Ordena las actividades en fases sucesivas —especificación, diseño, implementación, verificación y mantenimiento— y exige cerrar cada una antes de abrir la siguiente. Su ventaja es la trazabilidad: cada fase deja documentos revisables, lo que encaja bien en desarrollos con requisitos contractuales o sujetos a certificación.

Su inconveniente es el que lo hizo célebre: los requisitos rara vez se conocen por completo al principio, y el modelo hace caro cambiarlos. Un error de especificación no se detecta hasta la validación, cuando corregirlo cuesta órdenes de magnitud más que en su fase de origen [2].

### Desarrollo incremental

Divide el sistema en incrementos que se especifican, construyen y entregan por separado, empezando por los de mayor valor. Cada entrega es funcional y sirve para obtener realimentación antes de comprometerse con el resto.

Reduce el coste de acomodar cambios y adelanta el momento en que el cliente ve algo que funciona. A cambio, el proceso es menos visible desde fuera —no hay documentos de fase que marquen el avance— y la estructura del sistema tiende a degradarse si no se reserva esfuerzo para reorganizarla.

### Modelos evolutivos y por prototipos

Cuando el problema no está claro, resulta razonable construir una versión reducida que sirva para entenderlo. El prototipo puede ser *desechable*, si su único fin es aclarar requisitos y luego se descarta, o *evolutivo*, si se refina hasta convertirse en el producto. El riesgo del segundo es conocido: un prototipo se construye sin las garantías que se exigen a un producto, y promoverlo tal cual deja una base frágil.

El **modelo en espiral** de Boehm organiza el desarrollo en ciclos y sitúa el análisis de riesgos como actividad explícita al principio de cada uno. Su aportación duradera no es la forma de la espiral, sino la idea de que el proceso debe adaptarse al riesgo asumido.

### Métodos ágiles

Los métodos ágiles parten de una constatación: en muchos proyectos el coste de predecir es mayor que el de adaptarse. Priorizan las entregas frecuentes, la colaboración con el cliente y la respuesta al cambio por encima de la planificación detallada y la documentación exhaustiva. Scrum organiza el trabajo en iteraciones cortas con roles y reuniones definidos; la programación extrema añade prácticas técnicas como la integración continua, la programación por parejas y el desarrollo guiado por pruebas [6, 7].

No son la respuesta a todo. Su eficacia se apoya en equipos pequeños, en un cliente disponible y en un dominio donde equivocarse sea barato. En sistemas con certificación obligatoria, con muchos equipos coordinados o con un coste de fallo muy alto, conviene un proceso más ceremonioso, aunque adopte prácticas ágiles en lo técnico.

# Capítulo 2

# Ingeniería de requisitos

## 2.1. Introducción a la ingeniería de requisitos

Determinar qué debe hacer un sistema es más difícil que construirlo. El problema no es técnico: consiste en enunciar con precisión un problema complejo del que las personas implicadas tienen ideas distintas, a veces incompatibles y casi siempre incompletas.

Un **requisito** es una condición o capacidad que el sistema debe satisfacer. La palabra se usa con dos niveles de detalle que conviene no mezclar [1]:

- Los **requisitos de usuario** se enuncian en lenguaje natural, desde el punto de vista de quien va a usar el sistema, y sirven para acordar el alcance con el cliente.
- Los **requisitos del sistema** detallan las funciones, servicios y restricciones con la precisión necesaria para diseñar y contratar.

La distinción clásica es otra, y es transversal a la anterior:

- **Requisitos funcionales**: lo que el sistema debe hacer.
- **Requisitos no funcionales**: las restricciones sobre cómo debe hacerlo —rendimiento, disponibilidad, seguridad, normativa aplicable—.

Los no funcionales suelen recibir menos atención y son los que más condicionan la arquitectura. Un requisito de tiempo de respuesta o de tolerancia a fallos afecta al sistema entero, mientras que un requisito funcional suele quedar confinado a un módulo. Además, incumplir un requisito no funcional puede inutilizar el sistema aunque todas sus funciones estén implementadas.

### Propiedades de un buen conjunto de requisitos

Un requisito bien escrito es *necesario*, *verificable*, *no ambiguo* y *consistente* con los demás. La verificabilidad es la propiedad que más se descuida: «el sistema debe ser rápido» no es un requisito, porque no existe prueba que decida si se cumple. «El sistema debe responder a una consulta de catálogo en menos de dos segundos con cien usuarios concurrentes» sí lo es.

Al conjunto se le exige además que sea *completo* y *trazable*: que cubra todo lo necesario y que cada elemento del diseño y de las pruebas pueda remontarse

al requisito que lo justifica. La completitud absoluta es inalcanzable en sistemas grandes, y esa es una de las razones por las que los procesos iterativos ganaron terreno.

#### Delimitar lo que queda fuera

Tan importante como decir qué hace el sistema es dejar por escrito qué no hace. Un alcance que no se cierra crece durante el desarrollo sin que nadie lo decida, y ese crecimiento es una de las causas habituales de que un proyecto se desvíe en plazo y coste. La lista de exclusiones debe ser explícita y estar acordada.

## 2.2. Obtención de requisitos

La obtención —o elicitación— es la actividad de descubrir los requisitos hablando con las personas implicadas y observando cómo trabajan. Las dificultades son recurrentes: quienes van a usar el sistema describen su trabajo en términos de la herramienta que ya conocen, dan por supuesto lo que para ellos es evidente, y distintos grupos tienen objetivos que chocan.

Las técnicas más usadas son:

- **Entrevistas**, abiertas o estructuradas. Son el instrumento principal. Las abiertas sirven para explorar; las estructuradas, para confirmar y precisar.
- **Observación** del trabajo real. Descubre lo que nadie menciona en una entrevista porque lo hace de forma automática, y también los atajos con los que se sortea el sistema actual.
- **Talleres** con varios grupos a la vez, útiles para sacar a la luz los conflictos entre ellos en vez de descubrirlos al final.
- **Prototipos** y maquetas, que convierten una discusión abstracta en una reacción concreta ante algo visible.
- **Estudio de la documentación** existente y del sistema al que se sustituye.

Identificar a los *stakeholders* —toda persona u organización afectada por el sistema— forma parte de la actividad. Olvidar a un grupo es la vía más directa a un requisito descubierto tarde: los administradores del sistema, el personal de soporte y los responsables de cumplimiento normativo quedan fuera de la lista con frecuencia y aportan restricciones que luego resultan innegociables.

## 2.3. Modelado de casos de uso e historias de usuario

#### Casos de uso

Un caso de uso describe una interacción entre el sistema y un actor externo que produce un resultado con valor para ese actor. El **actor** es un rol, no una persona: el mismo empleado puede actuar como «cliente» y como «administrador» en casos distintos.

El diagrama de casos de uso de UML solo aporta la vista general: qué actores hay y con qué casos se relacionan. El contenido está en la descripción textual de cada caso, que suele incluir el actor principal, las precondiciones, el flujo principal paso a paso, los flujos alternativos y la postcondición [3, 4].

Las relaciones entre casos —`include` para el comportamiento común y `extend` para el opcional— conviene usarlas con moderación. Un modelo lleno de descomposiciones se vuelve ilegible y empieza a describir el diseño en vez del comportamiento observable, que es justo lo que un caso de uso debe evitar.

El error más frecuente al modelar es escribir los casos desde dentro del sistema, en términos de pantallas y de operaciones sobre la base de datos. Un caso de uso debe poder leerlo el cliente y reconocer su trabajo en él.

### Historias de usuario

Las historias de usuario son la alternativa ágil: enunciados breves, en la forma «como *rol* quiero *acción* para *beneficio*», acompañados de criterios de aceptación. No pretenden documentar el sistema, sino recordar una conversación pendiente con el cliente y servir de unidad de planificación.

Frente a los casos de uso son más ligeras y más fáciles de reordenar por prioridad, pero dejan menos rastro. La elección entre unos y otras no es de moda: depende de si el cliente está disponible de forma continua —caso en el que la conversación sustituye al documento— o de si el acuerdo tiene que sostenerse por sí solo durante meses.

## 2.4. Especificación y análisis

### Especificación

Especificar es dejar los requisitos por escrito de forma que sirvan de contrato técnico. El documento de especificación —el *SRS*, por sus siglas en inglés— recoge la introducción y el alcance, la descripción general del sistema y su entorno, los requisitos funcionales, los no funcionales y las restricciones, más los apéndices con glosario y modelos.

El estilo importa. Los requisitos se enuncian en frases cortas, una condición por requisito, con identificador propio y sin conjunciones que escondan dos exigencias en una. Las palabras que expresan obligación deben usarse con un significado acordado y constante, y conviene evitar los términos que no admiten prueba: «adecuado», «amigable», «eficiente», «si es posible».

### Análisis

El análisis transforma los requisitos en modelos que permiten razonar sobre ellos antes de diseñar. En un enfoque orientado a objetos, el resultado típico es el **modelo conceptual** o modelo del dominio: las clases del problema, sus atributos y las asociaciones entre ellas [3].

Ese modelo no es un diseño. Recoge los conceptos del dominio tal como existen para el cliente, sin métodos, sin decisiones de implementación y sin clases técnicas. Confundir ambas cosas produce un modelo del dominio contaminado de controladores y gestores, que ya no sirve para hablar con quien conoce el negocio.

Junto al modelo conceptual se emplean los **diagramas de secuencia del sistema**, que muestran las operaciones que un actor invoca sobre el sistema visto como una caja negra, y los **contratos de operación**, que precisan el efecto de cada una en términos de precondiciones y postcondiciones.

### Validación y gestión de los requisitos

Validar los requisitos es comprobar que describen el sistema que el cliente quiere, frente a verificar, que es comprobar que el sistema construido cumple los requisitos. Las técnicas habituales son las revisiones formales con los implicados, la construcción de prototipos y la redacción anticipada de los casos de prueba: un requisito para el que no se sabe escribir una prueba está mal enunciado.

Los requisitos cambian durante todo el proyecto, así que necesitan gestión: una línea base acordada, un procedimiento para proponer y aprobar cambios, y trazabilidad para saber qué se ve afectado por cada uno. Sin trazabilidad, el coste de un cambio se estima a ojo y siempre por debajo.

# Capítulo 3

# Diseño del software

## 3.1. Conceptos y principios de diseño

El diseño es la actividad que decide cómo se organiza el sistema para satisfacer los requisitos. Entre la especificación y el código hay un espacio de decisiones que no está determinado por ninguno de los dos, y es ahí donde se juega la mantenibilidad futura.

Los conceptos que sostienen esas decisiones son pocos y llevan décadas siendo los mismos [2]:

- **Abstracción.** Trabajar con una descripción que retiene lo relevante y omite el resto, y hacerlo por niveles.
- **Modularidad.** Dividir el sistema en partes con una responsabilidad identificable, de forma que cada una pueda entenderse y modificarse por separado.
- **Ocultación de información.** Que cada módulo esconda sus decisiones internas tras una interfaz. Es lo que permite cambiar la implementación sin arrastrar a los clientes del módulo.
- **Separación de intereses.** Que cada parte se ocupe de una cuestión y no de varias entremezcladas.

De ellos se derivan las dos medidas con las que se juzga una descomposición:

- La **cohesión** mide hasta qué punto los elementos de un módulo contribuyen a un mismo propósito. Interesa que sea alta.
- El **acoplamiento** mide la dependencia entre módulos. Interesa que sea bajo, y sobre todo que las dependencias vayan hacia abstracciones estables y no hacia detalles.

Las dos van juntas: una descomposición que reparte una misma responsabilidad entre varios módulos baja la cohesión y sube el acoplamiento a la vez, porque esos módulos tendrán que cambiar siempre juntos.

Por encima de estas medidas están los principios que guían la asignación de responsabilidades. Los patrones GRASP —experto en información, creador, controlador, bajo acoplamiento, alta cohesión— dan un vocabulario para justificar por qué una operación va en una clase y no en otra, que es la decisión más frecuente del diseño orientado a objetos [3].

## 3.2. Diseño de los casos de uso

El diseño de casos de uso conecta el análisis con la estructura de objetos: partiendo de las operaciones del sistema identificadas en el análisis, se decide qué objetos colaboran para realizarlas y con qué mensajes.

El instrumento habitual es el **diagrama de interacción**, en cualquiera de sus dos formas. El diagrama de secuencia ordena los mensajes en el tiempo y se lee bien cuando el flujo importa; el diagrama de comunicación destaca los enlaces entre objetos y se lee mejor cuando lo que interesa es la estructura de colaboración [4].

Diseñar una interacción obliga a repartir responsabilidades, y ese reparto es el que determina el acoplamiento del resultado. Dos criterios ayudan a decidirlo:

- Asignar cada responsabilidad a la clase que ya tiene la información necesaria para cumplirla, en vez de crear una clase que pida esa información a todas las demás.
- Introducir un *controlador* que reciba los eventos del sistema y delegue, en lugar de dejar que la capa de presentación hable directamente con el dominio.

Cuando un objeto necesita datos de muchos otros para hacer su trabajo, la interacción lo delata: el diagrama se llena de mensajes de consulta salientes. Esa forma es la señal de que la responsabilidad está mal colocada.

## 3.3. Diseño de la estructura de objetos

El resultado del diseño de las interacciones se consolida en el **diagrama de clases de diseño**, que añade al modelo conceptual lo que el análisis deliberadamente omitía: las operaciones, la visibilidad de los atributos, los tipos, la navegabilidad de las asociaciones y las clases que no pertenecen al dominio sino a la solución.

Hay que distinguirlo del modelo conceptual del que procede. Aquel describía el problema; este describe la solución, y por eso puede contener clases que no existen en el dominio —repositorios, adaptadores, fábricas— sin que eso sea un defecto.

Sobre esa estructura se aplican los **patrones de diseño**, soluciones probadas a problemas recurrentes. Los que más se emplean en este nivel son los de creación, para desacoplar al cliente de la clase concreta que instancia; los estructurales, como adaptador o fachada, para acomodar interfaces que no encajan; y los de comportamiento, como estrategia u observador, para dejar variable lo que se sabe que va a cambiar [3].

Los patrones se aplican cuando el problema que resuelven está presente. Aplicarlos por anticipado, «por si acaso», añade indirección sin beneficio y es una forma frecuente de complicar un diseño sin necesidad.

## 3.4. Arquitectura del software

La arquitectura es el conjunto de decisiones de diseño de mayor alcance: la descomposición del sistema en partes principales, la relación entre ellas y las propiedades globales que esa organización garantiza. Son las decisiones más caras de revertir, porque el resto del diseño se apoya en ellas.

La arquitectura la determinan, sobre todo, los requisitos no funcionales. El rendimiento empuja hacia componentes gruesos y poca comunicación entre ellos; la seguridad, hacia capas con control de acceso en las fronteras; la disponibilidad, hacia la redundancia; la mantenibilidad, hacia componentes pequeños y reemplazables. Como estas fuerzas tiran en direcciones opuestas, no existe una arquitectura buena en abstracto, solo una adecuada a unas prioridades declaradas [1].

Los estilos más comunes son:

- **En capas.** Cada capa usa solo servicios de la inmediatamente inferior. Facilita sustituir una capa entera y aísla el dominio de la tecnología, a costa de rendimiento en las llamadas que la atraviesan.
- **Cliente-servidor.** Separa a quien pide de quien sirve, lo que permite escalar cada lado por separado.
- **Modelo-vista-controlador.** Aísla la presentación del estado y de la lógica, de modo que varias vistas puedan compartir un mismo modelo.
- **Por componentes o servicios.** Divide el sistema en unidades desplegables de forma independiente. Gana en escalabilidad y en autonomía de los equipos; paga con complejidad de operación y con los problemas propios de la comunicación en red.
- **Tuberías y filtros.** Encadena transformaciones sobre un flujo de datos. Encaja bien en procesamiento por lotes.

Decidir la arquitectura incluye documentarla. Una arquitectura que solo existe en la cabeza de quien la diseñó deja de cumplirse en cuanto el equipo crece, y la erosión resultante es difícil de revertir porque el código ya depende de la forma degradada.

# Capítulo 4

# Otros aspectos de la Ingeniería del Software

## 4.1. Planificación y gestión de proyectos software

Gestionar un proyecto software consiste en conseguir que se entregue en plazo, dentro del presupuesto y con la calidad acordada. Es una actividad de ingeniería tanto como el diseño, y comparte con él la dificultad de decidir con información incompleta.

Los proyectos software tienen rasgos que los distinguen de otros: el producto es intangible, así que el avance no se ve; no hay dos proyectos iguales, de modo que la experiencia previa se traslada mal; y el proceso cambia con la tecnología a un ritmo que envejece rápido los datos históricos [1].

### Estimación

Estimar es predecir esfuerzo, plazo y coste. Los enfoques habituales son tres, y suelen combinarse:

- **Juicio de expertos**, con varias personas estimando por separado y contrastando después. Es barato y sorprendentemente robusto cuando el equipo conoce el dominio.
- **Estimación por analogía**, comparando con proyectos anteriores medidos.
- **Modelos algorítmicos**, como COCOMO, que calculan el esfuerzo a partir de una medida del tamaño y de factores de ajuste.

Todos comparten el mismo punto débil: dependen de una estimación del tamaño que se hace cuando menos se sabe del sistema. De ahí la práctica de estimar por rangos, revisar la estimación en cada iteración y desconfiar de las cifras cerradas dadas antes de la especificación.

Conviene recordar además que el esfuerzo y el plazo no son intercambiables. Añadir personas a un proyecto retrasado aumenta el coste de coordinación y suele retrasarlo más, observación que Brooks formuló hace medio siglo y que sigue describiendo bien lo que ocurre.

### Planificación y riesgos

La planificación descompone el trabajo en tareas con dependencias, asigna recursos y fija hitos. Los hitos deben corresponder a entregables verificables; un hito que se declara cumplido porque «está casi terminado» no informa de nada.

La gestión de riesgos es la parte de la planificación que se salta con más frecuencia y la que más pronto se echa en falta. Consiste en identificar lo que puede salir mal, estimar su probabilidad y su impacto, decidir qué hacer con cada riesgo —evitarlo, mitigarlo, transferirlo o aceptarlo— y revisar la lista de forma periódica. Los riesgos habituales en software son la rotación del personal, el cambio de requisitos, la subestimación del tamaño y la dependencia de tecnología o proveedores no probados.

### Calidad y métricas

La gestión de la calidad establece qué se entiende por calidad en el proyecto y cómo se comprueba: estándares de producto, estándares de proceso, revisiones e inspecciones. Las métricas —tamaño, complejidad, densidad de defectos, cobertura de pruebas— sirven para tomar decisiones, no para evaluar personas. Usadas como objetivo, dejan de medir lo que medían, porque el equipo optimiza el indicador y no la propiedad que representaba.

## 4.2. Validación y verificación de software

Verificar es comprobar que el sistema se construye conforme a su especificación; validar es comprobar que la especificación describe lo que el cliente necesita. La distinción clásica lo resume bien: la verificación pregunta si se está construyendo el producto correctamente, y la validación, si se está construyendo el producto correcto.

### Pruebas

Las pruebas ejecutan el sistema con la intención de encontrar defectos. Un resultado sin fallos no demuestra que no los haya: las pruebas muestran la presencia de errores, nunca su ausencia. Se organizan en niveles:

- **Unitarias**, sobre componentes aislados, escritas normalmente por quien los implementa y automatizadas.
- **De integración**, sobre las interacciones entre componentes, donde aparecen los fallos de interfaz que las unitarias no ven.
- **De sistema**, sobre el producto completo, incluidos los requisitos no funcionales: carga, seguridad, recuperación.
- **De aceptación**, ejecutadas con el cliente o en sus términos, y que pertenecen a la validación más que a la verificación.

El diseño de los casos de prueba puede partir de la especificación —prueba de caja negra, con técnicas como las clases de equivalencia y el análisis de valores límite— o de la estructura interna —prueba de caja blanca, guiada por criterios

de cobertura—. Ninguno de los dos basta por sí solo: la caja negra no llega a los caminos no previstos y la caja blanca no detecta lo que falta implementar [5].

La **prueba de regresión** es la que se vuelve a pasar tras cada cambio para comprobar que no se ha roto lo que funcionaba. Solo es viable si está automatizada, y por eso la automatización de pruebas es una decisión de proceso más que una preferencia técnica. En la programación extrema se lleva al extremo escribiendo la prueba antes que el código [7].

#### Revisiones e inspecciones

No toda verificación requiere ejecutar. Las revisiones e inspecciones examinan documentos y código con una lista de comprobación, y encuentran defectos que las pruebas no alcanzan, entre ellos los de la propia especificación. Su ventaja es que actúan antes: un defecto detectado en la revisión de requisitos cuesta una fracción de lo que cuesta corregido en producción.

## 4.3. Mantenimiento de software

El mantenimiento es todo el trabajo que se hace sobre el sistema después de la entrega, y es la fase más larga y más cara del ciclo de vida. Se clasifica en cuatro tipos:

- **Correctivo**, para reparar defectos.
- **Adaptativo**, para acomodar cambios del entorno: sistema operativo, normativa, sistemas con los que se integra.
- **Perfectivo**, para añadir funcionalidad o mejorar el rendimiento a petición del usuario.
- **Preventivo**, para facilitar el mantenimiento futuro sin cambiar el comportamiento observable.

Contra la intuición, el correctivo es la parte menor. La mayoría del esfuerzo se va en perfectivo y adaptativo, es decir, en hacer evolucionar un sistema que funciona [1].

#### Evolución y deterioro

Las leyes de Lehman describen lo que le ocurre a un sistema en uso: si no cambia, deja de resultar útil, porque el entorno sí cambia; y a medida que cambia, su complejidad crece salvo que se dedique trabajo explícito a contenerla. De ahí que el mantenimiento preventivo —refactorizar, actualizar dependencias, recuperar la documentación— no sea un lujo, sino la condición para que los otros tres tipos sigan siendo asequibles.

La **deuda técnica** nombra ese fenómeno desde el lado económico: cada decisión tomada por rapidez en lugar de por calidad genera un interés que se paga en cada cambio posterior. Es legítimo contraerla de forma consciente y con un plan para devolverla; el problema aparece cuando se contrae sin registrarla, porque entonces nadie decide cuándo se paga.

### Reingeniería

Cuando el coste de mantener supera al de rehacer, cabe la reingeniería: reconstruir el sistema conservando su funcionalidad. Es menos arriesgada que un desarrollo desde cero, porque el sistema existente documenta el comportamiento esperado, incluido el que nadie llegó a escribir en ningún requisito. La alternativa de reescribir de cero pierde ese conocimiento y suele reintroducir defectos que ya estaban resueltos.

# Bibliografía


[1] I. Sommerville, *Software Engineering*, 10ª ed., Pearson, 2016.

[2] R. S. Pressman, *Ingeniería del Software: un enfoque práctico*, McGraw-Hill, 2021.

[3] C. Larman, *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*, 3ª ed., Pearson, 2006.

[4] J. Arlow e I. Neustadt, *UML 2*, Anaya Multimedia, 2006.

[5] S. L. Pfleeger, *Ingeniería de Software: teoría y práctica*, Prentice Hall, 2002.

[6] C. Lasa Gómez, A. Álvarez García y R. de las Heras del Dedo, *Métodos Ágiles: Scrum, Kanban, Lean*, Anaya Multimedia, 2017.

[7] K. Beck, *Una explicación de la Programación Extrema: aceptar el cambio*, Addison-Wesley, 2002.