Compás
capacitación e investigación

2025

De la Idea al Producto:
Ingeniería de Software Paso a Paso

John Patricio Orellana Preciado

De la Idea al Producto:
Ingeniería de Software Paso a Paso

John Patricio Orellana Preciado

ISBN: 978-9942-53-116-2 DOI: http://doi.org/10.48190/9789942531162

Compás
capacitación e investigación

Primera edición, 11/11/2025

ISBN: 978-9942-53-116-2
DOI: http://doi.org/10.48190/9789942531162
Distribución online
Acceso abierto

Cita
Orellana, J. (2025) De la Idea al Producto: Ingeniería de Software Paso a Paso. Editorial Grupo Compás

Este libro es parte de la colección de la Univesidad Técnica de Machala y ha sido debidamente examinado y valorado en la modalidad doble par ciego con fin de garantizar la calidad de la publicación.

Introducción

¿Es posible convertir una simple idea en una solución tecnológica funcional, escalable y de calidad, sin perderse en la complejidad de los procesos de desarrollo? Esta pregunta representa uno de los mayores desafíos tanto para estudiantes como para profesionales que se inician en la Ingeniería de Software. En un mundo donde el software permea todos los sectores, desde la educación y la salud hasta la industria y el entretenimiento, comprender cómo se diseña, planifica y construye un producto de software se vuelve una competencia esencial.

A pesar del avance vertiginoso de las tecnologías, muchos proyectos de software fracasan o presentan deficiencias porque no se comprende a fondo el proceso ingenieril que hay detrás de su construcción. En el ámbito académico, esto se traduce en una necesidad urgente: formar profesionales capaces de abordar el desarrollo de software no solo desde la codificación, sino desde una visión integral que incluya planificación, análisis de requerimientos, modelado y diseño.

Este libro responde a esa necesidad. Su enfoque académico y práctico busca cerrar la brecha entre la teoría impartida en el aula y las competencias que exige la industria. Está concebido como un texto de asignatura para estudiantes de carreras tecnológicas, especialmente en el área de Tecnologías de la Información, aunque también resulta útil para docentes, investigadores y profesionales que desean actualizarse en metodologías modernas de desarrollo.

Numerosos autores han abordado la Ingeniería de Software desde diversas perspectivas. Pressman y Maxim (2021) han proporcionado marcos teóricos sólidos; Sommerville (2016) ha enfatizado la importancia del proceso como eje organizador; otros textos han centrado su atención en metodologías ágiles (como Scrum y XP), la gestión de requisitos o el diseño orientado a objetos. Sin embargo, la mayoría de estas obras presentan contenidos fragmentados o muy técnicos, lo que dificulta su aplicación directa en la enseñanza universitaria. Este libro busca ser un puente:

combina los fundamentos teóricos con recursos pedagógicos claros y aplicables en el aula.

El objetivo principal de este texto es guiar al lector a lo largo de las etapas fundamentales del desarrollo de software, desde la concepción de la idea hasta el diseño detallado del producto. El lector podrá comprender los principios de la Ingeniería de Software, familiarizarse con modelos de desarrollo (incluyendo los modelos ágiles), aprender a recolectar y modelar requisitos, y aplicar conceptos clave en el diseño estructurado y orientado a objetos.

Para alcanzar este propósito, el contenido ha sido organizado en cuatro capítulos temáticos que siguen una secuencia lógica y didáctica. Cada capítulo incluye actividades, esquemas, ejercicios, casos de estudio y recursos gráficos que promueven un aprendizaje activo y significativo. La metodología utilizada se basa en la investigación bibliográfica, el análisis comparativo de modelos y metodologías, y la integración de experiencias prácticas en entornos educativos.

En suma, este libro no solo enseña a desarrollar software; enseña a pensar como ingeniero de software. Es una invitación a descubrir el proceso creativo, técnico y metodológico que permite transformar una necesidad en una solución tecnológica efectiva.

Objetivo General

El propósito principal de *la Idea al Producto: Ingeniería de Software Paso a Paso* es ofrecer un enfoque integral que combine fundamentos teóricos sólidos con aplicaciones prácticas, orientado tanto al ámbito académico como a las necesidades reales de la industria del software. Este texto está diseñado como una guía formativa para docentes y estudiantes universitarios de carreras tecnológicas, así como un recurso de actualización profesional para quienes desean profundizar en el proceso sistemático de desarrollo de productos de software.

El contenido del libro abarca los pilares esenciales de la Ingeniería de Software, organizados en torno a los siguientes temas principales: introducción a la disciplina y sus fundamentos, modelos de proceso con especial énfasis en el desarrollo ágil, aspectos humanos que influyen en la producción de software como la usabilidad y el trabajo colaborativo, principios que orientan la práctica de la ingeniería, recolección y análisis de requerimientos, modelado del software en distintos niveles (estructural, conductual y de datos) y diseño de soluciones apoyado en patrones arquitectónicos y de componentes.

A lo largo del texto, se promueve un enfoque práctico y contextualizado, mediante el uso de estudios de caso que permiten aplicar los conocimientos adquiridos en dominios específicos como la salud, la seguridad informática, los sistemas financieros, la automatización de procesos y el comercio electrónico. Estos casos no solo refuerzan los conceptos abordados, sino que también vinculan la teoría con escenarios reales de desarrollo y gestión de software.

Al finalizar la lectura y estudio de este libro, los lectores estarán en capacidad de:

- Comprender los principios fundamentales que rigen la Ingeniería de Software moderna.
- Analizar necesidades del usuario y del entorno para la definición de requerimientos claros y viables.
- Modelar soluciones funcionales a través de herramientas, lenguajes y representaciones gráficas estándar.
- Diseñar sistemas de software aplicando conceptos de modularidad, reutilización y patrones de diseño.
- Reflexionar sobre el papel de los equipos humanos, la ética profesional y la calidad del software en entornos reales de trabajo.

En resumen, este libro proporciona las bases necesarias para que los futuros profesionales del software puedan

transformar una idea inicial en un producto útil, eficiente y alineado con estándares académicos y las exigencias del mercado.

Estructura del libro

Este libro ha sido diseñado con una organización lógica, progresiva y pedagógica que permite a los lectores avanzar paso a paso desde los conceptos fundamentales de la Ingeniería de Software hasta su aplicación práctica en el diseño de soluciones reales. La estructura del texto responde a criterios académicos de secuenciación didáctica, y se encuentra dividida en cuatro capítulos temáticos interrelacionados.

Cada capítulo aborda un eje central del desarrollo de software, integrando fundamentos teóricos, reflexiones críticas y aplicaciones prácticas en distintos dominios. Los capítulos siguen un esquema común que facilita el aprendizaje:

- Elementos de apertura, como introducciones, objetivos, esquemas conceptuales y preguntas de enfoque, que preparan al lector para los contenidos de la unidad.
- Recursos pedagógicos internos, tales como ilustraciones, tablas, diagramas, ejercicios breves, preguntas de reflexión, los cuales enriquecen la comprensión del contenido y permiten su aplicación inmediata.
- Elementos de cierre, que refuerzan el aprendizaje mediante resúmenes, preguntas de revisión, actividades prácticas, autoevaluaciones y tareas orientadas a la aplicación del conocimiento en contextos reales.

A continuación, se describe brevemente el contenido de cada capítulo:

- Capítulo 1: Introducción a la Ingeniería de Software

Se presentan los fundamentos de la disciplina, el proceso del software, los costos asociados al desarrollo, los atributos de calidad, los retos actuales y los aspectos humanos que influyen en el éxito de un proyecto, como la usabilidad y el trabajo en equipo.

- Capítulo 2: Modelos Ágiles

Se exploran los modelos de desarrollo ágil, sus principios, beneficios y costos, así como su aplicabilidad en distintos entornos industriales. Se incluyen casos de uso que permiten comprender cómo se implementan estas metodologías en escenarios reales.

- Capítulo 3: Ingeniería de Requisitos y Modelado de Software

Se estudian los procesos de indagación, análisis, validación y modelado de requerimientos, utilizando técnicas modernas y diagramas UML. También se abordan los distintos enfoques de modelado (estructural, de clases, de flujo y de comportamiento), incluyendo herramientas aplicables a webapps.

- Capítulo 4: Diseño del Software

Se abordan los principios de diseño como abstracción, modularidad y ocultamiento de información, así como el diseño arquitectónico, patrones de diseño, componentes e interfaces. Este capítulo incluye un estudio de caso integrador que permite aplicar los conocimientos adquiridos.

La coherencia en el uso de recursos pedagógicos, la secuencialidad de los temas y el enfoque práctico de cada unidad hacen de este libro una herramienta útil tanto para la enseñanza universitaria como para la formación continua en entornos profesionales.

Características pedagógicas del libro

Este libro ha sido diseñado como un recurso académico orientado a facilitar el aprendizaje progresivo de los principios, procesos y prácticas fundamentales de la

Ingeniería de Software. Su enfoque didáctico busca guiar a los estudiantes y profesionales en la comprensión y aplicación de los contenidos clave de la disciplina, integrando teoría, práctica y reflexión.

Las principales características pedagógicas del texto incluyen:

- Organización temática clara y secuencial: El contenido está distribuido en capítulos que abordan los temas fundamentales de la Ingeniería de Software, desde sus bases conceptuales hasta el diseño de soluciones, siguiendo una lógica de complejidad creciente.
- Estructura uniforme en cada capítulo: Todos los capítulos incluyen elementos de apertura (como objetivos, introducciones y preguntas iniciales), recursos pedagógicos internos (como gráficos, tablas y ejercicios breves) y elementos de cierre (como resúmenes, actividades de aplicación, autoevaluaciones y preguntas de revisión), lo cual facilita la navegación y comprensión del contenido.
- Integración de teoría y práctica: A lo largo del libro se presentan ejemplos contextualizados, estudios de caso y ejercicios prácticos que permiten aplicar los conceptos abordados en situaciones reales o simuladas.
- Estilo técnico y accesible: El lenguaje empleado combina rigor académico con claridad expositiva, lo que permite su uso tanto en contextos de formación inicial como en procesos de actualización profesional.
- Material complementario visual: Se incluyen diagramas, ilustraciones y tablas que enriquecen la comprensión de los temas y permiten una mejor visualización de los procesos y modelos explicados.
- Autoevaluación y consolidación del aprendizaje: Los ejercicios de cierre al final de cada capítulo ofrecen al lector la posibilidad de comprobar sus avances y reforzar los conocimientos adquiridos.

Estas características hacen de este libro una herramienta pedagógica útil, funcional y adaptable a diferentes entornos de enseñanza-aprendizaje, ideal para su uso en cursos universitarios de Ingeniería de Software

Tabla de contenidos

Capítulo 1: Introducción a la ingeniería de software

Introducción del capítulo

En el mundo moderno, el software es parte fundamental de casi todos los aspectos de la vida diaria: desde los sistemas bancarios que gestionan nuestras finanzas hasta las aplicaciones móviles que usamos para comunicarnos. Sin embargo, desarrollar software útil, eficiente y confiable no es una tarea improvisada; requiere de principios, procesos y metodologías que forman parte de una disciplina estructurada: la Ingeniería de Software.

Este capítulo introduce los fundamentos esenciales de esta área, abordando qué es la Ingeniería de Software, cómo se organizan sus procesos, cuáles son sus costos y métodos más comunes, y qué atributos debe tener un buen software. También se examinan los principales desafíos que enfrenta la disciplina en la actualidad y se analizan aspectos humanos clave, como la importancia de la usabilidad y el trabajo en equipo dentro de los proyectos de desarrollo.

Objetivos de aprendizaje

Al finalizar el estudio de este capítulo, el lector será capaz de:

- Comprender qué es la Ingeniería de Software y su rol en el desarrollo tecnológico actual.
- Identificar las fases principales del proceso del software y su propósito dentro del ciclo de vida del desarrollo.
- Reconocer los distintos tipos de costos asociados al desarrollo de software y los métodos empleados para su gestión.
- Analizar los atributos que definen la calidad de un producto software.
- Reflexionar sobre los retos contemporáneos que enfrenta la Ingeniería de Software como disciplina.

- Valorar el papel de los factores humanos, incluyendo la usabilidad y el trabajo en equipo, en la producción de software de calidad.

Esquema del capítulo

1.1 Fundamentos de la ingeniería de software
1.1.1 Capas de la ingeniería de software
1.2 El proceso del software
1.3 Costos y métodos del desarrollo de software
1.4 Atributos de un buen software
1.5 Retos fundamentales en la ingeniería de software
1.6 Aspectos Humanos en la Ingeniería de Software
1.6.1 Usabilidad en el desarrollo de software
1.6.2 Trabajo en equipo en el desarrollo de software

Preguntas guía

- ¿Por qué es necesario hablar de "ingeniería" en el desarrollo de software?
- ¿En qué se diferencia el proceso de creación de software respecto a otros productos de ingeniería?
- ¿Qué elementos definen la calidad de un software?
- ¿Qué desafíos técnicos, económicos y humanos enfrenta hoy la Ingeniería de Software?
- ¿Cómo influyen la usabilidad y el trabajo en equipo en el éxito de un proyecto?

1.1 Fundamentos de la ingeniería de software

¿Qué es el software?

El software es el conjunto de programas, procedimientos y documentación relacionados con el funcionamiento de un sistema informático. A diferencia del hardware —que se refiere a los componentes físicos de un sistema—, el software representa la lógica y funcionalidad que permite al usuario interactuar con la máquina y ejecutar tareas específicas.

Existen diversos tipos de software, cada uno con características y propósitos distintos:

- Software de sistema: Permite la gestión de los recursos del hardware. Ejemplos: sistemas operativos, controladores de dispositivos, BIOS.
- Software de aplicación: Satisface necesidades específicas de los usuarios finales. Ejemplos: procesadores de texto, navegadores web, sistemas de facturación.
- Software de ingeniería y ciencias: Apoya actividades científicas, de simulación o cálculo. Ejemplos: MATLAB, software de simulación térmica, CAD.
- Software embebido: Se integra en dispositivos de hardware con funciones específicas. Ejemplos: software en microondas, sistemas de control automotriz.
- Software de línea de productos: Conjunto de programas relacionados que comparten componentes. Ejemplo: suite Microsoft Office.
- Aplicaciones web: Accesibles vía navegador, ejecutadas en servidores remotos. Ejemplo: Google Drive, plataformas de banca en línea.
- Software de inteligencia artificial: Imita capacidades humanas como el razonamiento, aprendizaje y percepción. Ejemplo: asistentes virtuales, sistemas expertos.

El software es la base lógica que impulsa la funcionalidad tecnológica de casi todos los dispositivos modernos.

Con esta comprensión básica, se hace evidente que el desarrollo de software es una actividad crítica y compleja, que no puede dejarse al azar. De allí nace la necesidad de una disciplina estructurada: la Ingeniería de Software.

¿Qué es la Ingeniería de Software?

La Ingeniería de Software es una disciplina de la ingeniería dedicada al diseño, desarrollo, prueba, mantenimiento y gestión de sistemas de software. A diferencia del desarrollo

de software informal o artesanal, esta disciplina aplica principios sistemáticos, técnicas comprobadas y metodologías estructuradas para garantizar la construcción de software de alta calidad, escalable, mantenible y alineado a las necesidades del usuario.

Definición (IEEE, 1990): *"La Ingeniería de Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software."*

Esto implica que el desarrollo de software no debe limitarse únicamente a la programación o codificación, sino que debe comprender todas las fases del ciclo de vida de un producto: desde la recolección de requerimientos hasta su implementación y mantenimiento posterior.

¿Por qué se necesita una ingeniería para el software?

Durante las décadas de 1960 y 1970, la industria del software vivió lo que se conoce como la *crisis del software*: una etapa caracterizada por proyectos que excedían costos y plazos, productos de baja calidad, y una creciente complejidad técnica. Esta situación reveló la necesidad de establecer una disciplina formal que permitiera controlar el proceso de desarrollo, similar a lo que ocurría en otras ramas de la ingeniería.

Términos clave:

- Crisis del software: Problemas generalizados en la industria del software que demostraron la necesidad de una disciplina formal.
- Disciplina ingenieril: Aplicación de principios, normas y buenas prácticas para garantizar calidad y eficiencia.
- Principios básicos de la Ingeniería de Software
- Los fundamentos de la Ingeniería de Software se construyen sobre principios como:
- Comprensión del problema: Todo desarrollo inicia con la identificación precisa de las necesidades del usuario.

- Planificación estructurada: Se requiere un enfoque metodológico y organizado para gestionar tareas, recursos y riesgos.
- Calidad como eje central: La calidad no se evalúa solo al final, sino que debe integrarse en cada etapa del proceso.
- Evolución continua: El software cambia con el tiempo. Debe adaptarse a nuevas necesidades y contextos tecnológicos.

"La Ingeniería de Software no se trata solo de crear código, sino de crear valor a través de soluciones tecnológicas confiables, sostenibles y centradas en el usuario."

Ejemplo práctico

Imaginemos una aplicación móvil para la gestión de citas médicas. Un enfoque puramente técnico se limitaría a programar pantallas y funcionalidades. En cambio, desde la perspectiva de la Ingeniería de Software se consideran aspectos como:

- ¿Quiénes son los usuarios finales (médicos, pacientes, administrativos)?
- ¿Qué procesos deben modelarse (agenda, historial, notificaciones)?
- ¿Qué requisitos de seguridad y escalabilidad debe cumplir el sistema?

Referencias cruzadas

El lector encontrará en el Capítulo 3 un desarrollo detallado sobre la ingeniería de requerimientos, donde se explican las técnicas para identificar y validar necesidades del usuario. Asimismo, en el Capítulo 4 se abordarán los principios de diseño que permiten construir soluciones modulares y reutilizables.

1.1.1 Capas de la ingeniería de software

La Ingeniería de Software puede entenderse como una disciplina organizada en capas jerárquicas, donde cada nivel proporciona el soporte necesario al que está por encima de

él. Estas capas forman un modelo conceptual que permite visualizar los distintos elementos que intervienen en el desarrollo del software, desde los más abstractos hasta los más operativos.

"Las capas de la Ingeniería de Software no son etapas secuenciales, sino elementos integrados que coexisten y se refuerzan mutuamente durante el desarrollo del producto."

Las cuatro capas principales según Pressman y Maxim (2021) son:

- Compromiso con la calidad
- Proceso
- Métodos
- Herramientas

1. Compromiso con la calidad

Esta es la capa más externa y general. La calidad no es una etapa, sino una filosofía que debe permear todas las actividades del desarrollo. Involucra prácticas que aseguren que el software cumpla con las expectativas del cliente, funcione correctamente, y sea confiable, mantenible y usable.

La calidad en el software no debe evaluarse al final, sino construirse desde el inicio.

2. Proceso

El proceso de software actúa como el marco organizativo que guía el desarrollo. Define qué hacer, cuándo hacerlo, cómo hacerlo y quién debe hacerlo. Esta capa se aborda con mayor detalle en la sección 1.2 del capítulo.

3. Métodos

Aquí se ubican los enfoques técnicos y prácticos utilizados para desarrollar el software. Incluye actividades como:

- Análisis de requerimientos
- Diseño estructurado y orientado a objetos
- Pruebas de software
- Gestión de configuración
- Mantenimiento evolutivo

Cada método está diseñado para cumplir con uno o varios objetivos del proceso y garantizar la calidad del producto.

4. Herramientas

Las herramientas apoyan la automatización de tareas técnicas y administrativas dentro del proceso de software. Pueden ser:

- Herramientas CASE (Computer-Aided Software Engineering)
- Editores de código, compiladores
- Sistemas de control de versiones (Git)
- Entornos de desarrollo integrados (IDEs)

Ejemplo: Para una actividad de diseño orientado a objetos, un método puede ser el uso de diagramas de clases UML, y una herramienta sería un modelador como StarUML o Enterprise Architect.

Relación entre las capas: Estas capas no actúan de forma aislada, sino como un sistema integrado. El compromiso con la calidad guía todas las decisiones, el proceso estructura las actividades, los métodos especifican cómo se realiza el trabajo técnico, y las herramientas facilitan su ejecución.

1.2 El proceso del software

¿Qué entendemos por proceso de software?

Un proceso de software es un conjunto estructurado de actividades que se llevan a cabo para especificar, diseñar, construir, probar, desplegar y mantener un sistema de software. Es la "hoja de ruta" que guía a los equipos de desarrollo desde la concepción de la idea hasta la entrega del producto final.

"El proceso de software proporciona el marco necesario para transformar una necesidad en una solución tecnológica operativa."

Este proceso no es uniforme ni único; puede variar dependiendo del tipo de proyecto, del enfoque metodológico (clásico o ágil), del tamaño del equipo y de los requisitos del cliente.

Actividades fundamentales del proceso de software

Según Pressman y Maxim (2021), todo proceso de software bien definido incluye, al menos, las siguientes actividades genéricas:

1. Comunicación: Requiere establecer contacto con el cliente o usuario final para entender sus necesidades y expectativas.
2. Planeación: Implica definir el trabajo, estimar tiempo y recursos, y planificar actividades.
3. Modelado: Consiste en analizar los requerimientos y representar el sistema mediante modelos conceptuales.
4. Construcción: Incluye la codificación, integración y pruebas iniciales del software.
5. Entrega y retroalimentación: El producto se entrega al usuario, quien lo evalúa y genera comentarios para futuras mejoras.
6. Soporte y mantenimiento: Implica corregir errores, adaptar el software a nuevos entornos y mejorar funcionalidades.

Término clave: Ciclo de vida del software – Conjunto de fases por las que atraviesa un producto software desde su concepción hasta su retiro.

Modelos de proceso: una representación del camino

El proceso de software puede adoptar diferentes modelos organizativos, conocidos como modelos de proceso. Estos modelos ofrecen una guía estructurada para planificar, organizar y controlar las actividades de desarrollo. Aunque en los últimos años los enfoques ágiles han ganado popularidad, los modelos tradicionales siguen siendo fundamentales para comprender las bases metodológicas del desarrollo de software.

Según Sommerville (2016) y Pressman y Maxim (2021), los modelos tradicionales proporcionan estructura, control y documentación formal, siendo adecuados especialmente en

proyectos donde los requisitos son bien conocidos desde el principio.

Entre los modelos tradicionales más destacados se encuentran:

Modelo en Cascada

El modelo en cascada fue uno de los primeros enfoques propuestos formalmente para el desarrollo de software (Royce, 1970). Propone un proceso secuencial, en el que cada fase debe completarse antes de comenzar la siguiente.

Fases principales:

- Análisis de requisitos
- Diseño del sistema y del software
- Implementación (codificación)
- Prueba (verificación)
- Mantenimiento

Ventajas:

- Claridad en la documentación.
- Control estricto de avance y cumplimiento de fases.
- Ideal en proyectos con requisitos estables y bien definidos.
- Limitaciones:
- Rigidez: no es adecuado para entornos de cambios frecuentes.
- Difícil de adaptarse a nuevos requisitos descubiertos durante el desarrollo.
- La retroalimentación temprana del usuario es limitada.
- Sommerville (2016) destaca que el modelo en cascada es mejor utilizado en proyectos de sistemas críticos o embebidos donde la seguridad y verificación estricta son esenciales.

Modelo Incremental

El modelo incremental combina la estructura planificada del modelo en cascada con una estrategia iterativa de entrega. El

producto se desarrolla en una serie de incrementos, cada uno de los cuales aporta una funcionalidad parcial pero operativa.

Características principales:

- Cada incremento incorpora funcionalidades adicionales.
- El cliente puede utilizar las versiones preliminares mientras el sistema se desarrolla completamente.
- Ventajas:
- Reducción del riesgo: problemas pueden detectarse temprano.
- Flexibilidad ante cambios en los requisitos.
- Valor de negocio más rápido gracias a entregas tempranas.

Limitaciones**:**

- Puede ser difícil gestionar versiones intermedias si no se planifica adecuadamente.
- La arquitectura debe ser diseñada pensando en la expansión progresiva.

McConnell (2006) señala que el modelo incremental es eficaz para proyectos donde se requiere entrega rápida de funcionalidades clave para obtener retroalimentación temprana del usuario.

Modelo en Espiral

Propuesto por Barry Boehm en 1988, el modelo en espiral integra elementos de diseño iterativo y control de riesgos. El desarrollo avanza a través de espirales (ciclos) que incluyen actividades repetitivas de planificación, análisis de riesgos, construcción y evaluación.

Etapas en cada ciclo:

1. Establecimiento de objetivos.
2. Identificación y resolución de riesgos.
3. Desarrollo y prueba de la solución.
4. Planificación de la siguiente iteración.

Ventajas:

- Gestión proactiva de riesgos.
- Permite la evolución progresiva del sistema según necesidades.
- Adecuado para proyectos grandes y complejos.
- Limitaciones:
- Puede resultar costoso y complejo de gestionar.
- Requiere experiencia significativa en gestión de riesgos.

Según Boehm (1988), "no gestionar riesgos en proyectos grandes es una receta segura para el fracaso del software".

Modelos Ágiles (Mención preliminar)

Aunque los modelos ágiles como Scrum, XP y Kanban serán tratados en profundidad en el Capítulo 2, es importante señalar que surgen como una respuesta a las limitaciones de los modelos tradicionales. Promueven entregas rápidas, adaptabilidad al cambio, colaboración constante con el cliente y ciclos de desarrollo iterativos.

Beck et al. (2001), en el Manifiesto Ágil, enfatizan la importancia de "responder ante el cambio por encima de seguir un plan".

Tabla 1

Comparación entre modelos de desarrollo de software.

Modelo	**Ventajas**	**Limitaciones**	**Contexto ideal**
Cascada	Documentación formal, claridad	Rigidez, poca adaptabilidad	Requisitos fijos, sistemas críticos
Incremental	Flexibilidad, entregas tempranas	Gestión compleja de versiones	Proyectos donde se requiere valor temprano
Espiral	Gestión de riesgos, adaptabilidad	Costoso, complejo	Proyectos grandes, alta incertidumbre

Nota. Elaboración propia

Ejemplo ilustrativo

Supongamos que una empresa desea crear un sistema de gestión de inventarios. El proceso de software podría seguir estas etapas:

- En la fase de comunicación, se entrevistan a los encargados de bodega para identificar problemas actuales.
- En la planeación, se definen las funcionalidades clave (registro de productos, alertas de stock mínimo, reportes).
- En el modelado, se elaboran diagramas UML para representar el flujo del sistema.
- En la construcción, se desarrolla el backend, frontend y se integran las bases de datos.
- Tras la entrega, se realiza una prueba piloto con los usuarios, quienes dan retroalimentación.
- Luego, se pasa a la fase de mantenimiento, donde se resuelven incidencias y se ajusta el sistema a nuevos requerimientos.

Un proceso bien definido no solo mejora la calidad del producto, sino que también reduce costos, errores y retrabajos innecesarios

1.3 Costos y métodos del desarrollo de software

El costo del desarrollo de software representa el conjunto de recursos económicos, humanos y materiales necesarios para llevar un producto de software desde su concepción hasta su despliegue y mantenimiento. Estimar adecuadamente estos costos es fundamental para garantizar la viabilidad de un proyecto.

Según Ian Sommerville (2016), los factores que más impactan en el costo de un proyecto de software incluyen:

- Complejidad de los requerimientos: A mayor complejidad, mayor esfuerzo de desarrollo.
- Experiencia del equipo de desarrollo: Equipos más experimentados tienden a ser más eficientes.

- Tamaño del proyecto: El tamaño afecta la cantidad de recursos necesarios.
- Calidad requerida: Altos estándares de calidad requieren mayores inversiones en pruebas y validación.
- Tecnologías utilizadas: Tecnologías emergentes o desconocidas pueden incrementar costos de formación y adaptación.

DATO**:** Sommerville afirma que los costos de mantenimiento pueden representar entre el 60% y el 80% del costo total de vida de un software.

Steve McConnell (2006) en *Software Estimation: Demystifying the Black Art* resalta además que los costos de software están sujetos a una considerable incertidumbre, especialmente en las etapas tempranas de un proyecto, por lo que toda estimación debe verse como una predicción basada en la mejor información disponible en el momento.

Modelos de estimación de costos

Uno de los métodos clásicos más reconocidos es el Modelo COCOMO (Constructive Cost Model), desarrollado por Barry Boehm en 1981. Este modelo permite estimar el esfuerzo y tiempo de desarrollo basándose en el tamaño estimado del software (medido en líneas de código) y en ciertos factores de costo.

Tipos de COCOMO:

- COCOMO Básico: Proporciona una fórmula simple basada en el tamaño del software.
- COCOMO Intermedio: Añade factores de ajuste como confiabilidad requerida o complejidad del producto.
- COCOMO Detallado: Considera el costo de cada fase de desarrollo de manera separada.
- Ejemplo básico de fórmula COCOMO**:**
- $\text{Esfuerzo} = a \times (\text{KDSI})^b$

donde:

- a y b son constantes determinadas empíricamente,
- KDSI = miles de líneas de código fuente.

Barry Boehm resaltó que "sin una estimación inicial confiable, todo el planeamiento del proyecto está basado en supuestos riesgosos" (Boehm, 1981).

Ejemplo de COCOMO Básico

Planteamiento:

Una empresa de software debe desarrollar un sistema de gestión académica para una universidad.
Se estima que el sistema tendrá aproximadamente 50,000 líneas de código fuente (50 KLOC o 50KDSI).
El proyecto es de tipo orgánico (es decir, pequeño a mediano, con equipos bien organizados).
Para proyectos orgánicos, Barry Boehm (1981) propuso los siguientes valores típicos de los coeficientes:

- a=2.4
- b=1.05

La fórmula básica de esfuerzo es:

$$\text{Esfuerzo (en persona-meses)} = a \times (\text{KDSI})^b$$

Aplicación de la fórmula:

Datos:

- a=2.4
- b=1.05
- KDSI=50

Sustituimos:

$$\text{Esfuerzo} = 2.4 \times (50)^{1.05}$$

Primero, resolvemos

$$50^{1.05} = 56.234$$

Ahora multiplicamos:

$$\text{Esfuerzo} = 2.4 \times 56.234 = 134.9616$$

Resultado**:**

Esfuerzo estimado=135 persona-meses

Interpretación del resultado:

- El proyecto requerirá aproximadamente 135 persona-meses de esfuerzo.
- Esto significa, por ejemplo:
  - 10 personas trabajando durante 13.5 meses, o
  - 15 personas trabajando durante 9 meses.

(Claro que en la práctica también se consideran otras variables como eficiencia, tiempos muertos, y curvas de aprendizaje.)

Nota: En COCOMO, una "persona-mes" representa el trabajo realizado por una persona en un mes de dedicación completa.

Además de COCOMO, existen otros métodos de estimación relevantes:

- Puntos por función (Function Points) – Propuesto por Allan Albrecht, mide la funcionalidad ofrecida al usuario más que el tamaño del código.
- Métodos ágiles – En metodologías ágiles se emplean técnicas como planning poker y estimación por *story points*, que evalúan el esfuerzo relativo de las tareas de manera colaborativa.

Costos de errores

Uno de los aspectos más críticos en el desarrollo de software es que el costo de corregir errores crece exponencialmente cuanto más tarde son detectados. Según el estudio clásico de Barry Boehm, el costo de un error encontrado en la fase de mantenimiento puede ser hasta 100 veces mayor que si se hubiera detectado en la fase de requerimientos.

Importancia de la planificación del costo

Una correcta estimación de costos permite:

- Evaluar la viabilidad económica del proyecto.
- Planificar adecuadamente los recursos y tiempos de entrega.
- Identificar riesgos financieros de manera anticipada.
- Justificar ante stakeholders (clientes, gerentes, inversores) la inversión requerida.

"La planificación de costos no es solo una cuestión de presupuesto; es una estrategia fundamental para el éxito de cualquier proyecto de software" (McConnell, 2006).

1.4 Atributos de un buen software

Un buen software no solo cumple su propósito funcional; también debe satisfacer criterios de calidad que aseguren su utilidad, eficiencia, facilidad de uso y mantenimiento a largo plazo. Según diversos autores como Ian Sommerville (2016), Pressman y Maxim (2021), y normas internacionales como la ISO/IEC 25010, existen atributos esenciales que definen la calidad de un producto software.

"La calidad de un software no debe evaluarse únicamente por lo que hace, sino por cómo lo hace y qué tan bien lo sostiene en el tiempo."

A continuación, se describen los principales atributos:

1. Funcionalidad

La funcionalidad se refiere a que el software cumpla correctamente los requisitos para los cuales fue diseñado. Implica:

- Corrección: El software realiza las tareas esperadas de forma precisa.
- Adecuación: Proporciona un conjunto adecuado de funciones.
- Interoperabilidad: Se integra eficazmente con otros sistemas.
- Un sistema bancario que calcula incorrectamente los intereses carecería de funcionalidad, independientemente de su buena apariencia.

2. Confiabilidad

La confiabilidad mide la capacidad del software para mantener su nivel de servicio bajo condiciones específicas durante un período determinado.

Incluye:

- Disponibilidad: El software está operativo cuando se requiere.
- Tolerancia a fallos: Puede continuar funcionando ante ciertos errores.
- Capacidad de recuperación: Puede restablecerse después de fallos.

Sommerville (2016) resalta que en sistemas críticos —como control aéreo o dispositivos médicos— la confiabilidad es el atributo más importante.

3. Usabilidad

La usabilidad evalúa qué tan fácil es para los usuarios aprender a usar el sistema, operarlo eficientemente y obtener satisfacción con su uso.

Componentes:

- Facilidad de aprendizaje: Rápida comprensión inicial.
- Eficiencia de uso: Operaciones intuitivas y ágiles.
- Satisfacción del usuario: Experiencia positiva general.

Un software potente pero difícil de usar será abandonado por sus usuarios.

4. Eficiencia

La eficiencia mide el uso óptimo de los recursos del sistema (CPU, memoria, tiempo de respuesta) en la ejecución de sus funciones.

Indicadores:

- Rendimiento en tiempo de ejecución.
- Consumo de recursos.

Pressman y Maxim (2021) afirman que una aplicación ineficiente puede volverse inaceptable, incluso si su funcionalidad es correcta.

5. Mantenibilidad

La mantenibilidad se refiere a la facilidad con que el software puede ser modificado para corregir errores, adaptarse a cambios del entorno o mejorar su rendimiento.

Aspectos importantes:

- Modificabilidad: Facilidad para realizar cambios.
- Análisis: Facilidad para diagnosticar errores.
- Estabilidad: Poca probabilidad de efectos colaterales tras modificaciones.

La vida útil del software depende directamente de su mantenibilidad.

6. Portabilidad

La portabilidad mide la facilidad con la cual el software puede transferirse de un entorno operativo a otro.

Se evalúa en función de:

- Adaptabilidad: Capacidad de funcionar en diferentes plataformas.
- Instalabilidad: Facilidad de instalación en nuevos entornos.
- Compatibilidad: Funcionamiento correcto con hardware o software distinto.

ISO/IEC 25010 enfatiza que la portabilidad es clave en el mundo actual de múltiples dispositivos y arquitecturas.

Tabla 2

Atributos de un buen software

Atributo	**Descripción breve**
Funcionalidad	Cumple con los requisitos esperados.

Confiabilidad	Opera correctamente en diversas condiciones.
Usabilidad	Fácil de aprender y utilizar.
Eficiencia	Utiliza óptimamente los recursos.
Mantenibilidad	Fácil de corregir y mejorar.
Portabilidad	Funciona en distintos entornos.

Nota. Elaboración propia

En conclusión:

El diseño y construcción de software no deben enfocarse solo en hacer que funcione, sino en hacer que funcione bien, siempre, y en diferentes condiciones.
Estos atributos permiten construir productos que no solo cumplen objetivos funcionales, sino que generan confianza, satisfacción y valor sostenible para los usuarios y las organizaciones.

1.5 Retos fundamentales en la ingeniería de software

La Ingeniería de Software enfrenta una serie de retos persistentes que, a pesar de los avances tecnológicos, siguen afectando el éxito de los proyectos. Entender estos desafíos permite anticipar riesgos, planificar estrategias de mitigación y fortalecer la calidad de los productos desarrollados.

Esta visión se confirma en testimonios de profesionales activos. Por ejemplo, en una entrevista realizada a Laura Hernández, ingeniera de software senior en una empresa de soluciones financieras (Revista *Software Today*, 2023), destaca:

"Los principales retos que enfrentamos día a día no son tecnológicos, son humanos: cambios constantes de requerimientos, presión de plazos, coordinación entre equipos, y mantener la calidad en medio de todo eso. La

ingeniería de software es tan técnica como social." (Hernández, 2023).

Este testimonio evidencia que los desafíos en el desarrollo de software van más allá de las tecnologías utilizadas. Involucran la gestión de la incertidumbre, la colaboración multidisciplinaria y la habilidad para integrar calidad en todas las fases del ciclo de vida. Prepararse para afrontar estos retos es clave para el éxito en la práctica profesional.

Pressman y Maxim (2021) destacan que los principales retos pueden agruparse en varias categorías:

1. Naturaleza cambiante de los requisitos

Los requisitos de software rara vez se mantienen estáticos. A medida que los usuarios interactúan con prototipos o evolucionan sus necesidades, los requisitos tienden a cambiar.

- El reto consiste en gestionar los cambios de manera controlada para evitar desbordes de alcance ("scope creep") y garantizar la alineación del producto final con las expectativas actuales.

Cita destacada**:** *"El cambio es la única constante en el desarrollo de software."* —Pressman y Maxim, 2021.

2. Creciente complejidad de los sistemas

El tamaño y la interconexión de los sistemas modernos aumentan continuamente. El software actual debe integrarse con bases de datos masivas, dispositivos móviles, servicios en la nube, inteligencia artificial, entre otros.

- El desafío radica en diseñar arquitecturas escalables, modulares y sostenibles que soporten esta complejidad creciente sin comprometer la calidad.

Ian Sommerville (2016) señala que la complejidad accidental y esencial del software ha crecido exponencialmente, haciendo necesaria la adopción de buenas prácticas de ingeniería de requisitos, diseño y validación.

3. Alta demanda de calidad

Hoy, los usuarios no solo esperan que el software funcione, sino que también sea:

- Rápido
- Seguro
- Intuitivo
- Disponible 24/7
- Actualizable continuamente

El reto es integrar calidad desde el inicio del ciclo de vida del software, en lugar de tratarla como una fase final aislada.

McConnell (2004) afirma: "La calidad del software debe ser diseñada, no inspeccionada."

4. Presión de tiempo y costos

Los proyectos de software a menudo enfrentan restricciones de tiempo y presupuesto cada vez más estrictas, lo que puede forzar a los equipos a reducir fases críticas como la planificación, el diseño o las pruebas.

El desafío es lograr productos funcionales y de calidad optimizando el uso de recursos sin sacrificar procesos esenciales.

5. Gestión de equipos multidisciplinarios

El desarrollo de software moderno involucra a profesionales de diferentes disciplinas: programadores, analistas, diseñadores UX/UI, testers, especialistas en seguridad, entre otros.

- Coordinar equipos diversos y fomentar la colaboración efectiva representa un desafío en sí mismo.
- La comunicación fluida y el trabajo en equipo son factores determinantes para el éxito.

1.6 Aspectos Humanos de la Ingeniería de Software

La Ingeniería de Software no es solamente una actividad técnica; también es profundamente humana. Todo software es construido por personas para ser utilizado por otras personas. Esta realidad implica que tanto su creación como

su uso están inevitablemente condicionados por factores humanos: percepciones, emociones, capacidades cognitivas, dinámicas sociales y limitaciones individuales.

Pressman y Maxim (2021) insisten en que el factor humano debe considerarse en todas las etapas del ciclo de vida del software, no solo en la interfaz de usuario, sino también en la forma en que se gestionan equipos, se comunican requisitos y se abordan los cambios.

Cita destacada: "El software es más que código; es la representación lógica de una necesidad humana."-Pressman & Maxim, 2021.

Ian Sommerville (2016) también subraya que los problemas más serios en el desarrollo de software no provienen de la tecnología, sino de factores como la comunicación incompleta, los conflictos de intereses, la resistencia al cambio y las expectativas mal gestionadas.

El Human-Centered Design (Norman, 2013) introduce formalmente esta perspectiva: los sistemas tecnológicos deben diseñarse centrados en las personas, considerando sus capacidades, limitaciones y objetivos. Este enfoque reconoce que:

- Los errores humanos son inevitables, por lo que los sistemas deben ser tolerantes y comprensibles.
- Los usuarios no deben adaptarse al software; el software debe adaptarse a ellos.
- Las decisiones técnicas deben ser mediadas por el impacto que tienen en la experiencia del usuario.

Norman (2013): "La tecnología es desarrollada por seres humanos, para seres humanos, con todas las maravillas y defectos que eso implica."

De manera similar, Tom DeMarco y Timothy Lister en su célebre obra *Peopleware* (1987), argumentan que la productividad de los proyectos de software depende más de la gestión del talento humano que de las herramientas o metodologías técnicas. Factores como el ambiente de trabajo, la autonomía, la comunicación efectiva y la

motivación personal son determinantes para el éxito o el fracaso de los proyectos.

1.6.1 Usabilidad en el desarrollo de software

La usabilidad se refiere a la facilidad con la que los usuarios pueden aprender a utilizar un sistema, realizar tareas de manera eficiente y obtener una experiencia satisfactoria.

Según la norma ISO 9241-11, la usabilidad se mide a través de tres factores principales:

- Eficacia: El grado en que los usuarios logran sus objetivos.
- Eficiencia: El nivel de recursos utilizados en relación al logro de esos objetivos.
- Satisfacción: La percepción positiva del uso del sistema.
- Principios clave de la usabilidad (Nielsen, 1994):
- Visibilidad del estado del sistema: El sistema debe mantener informados a los usuarios sobre lo que está ocurriendo.
- Concordancia entre el sistema y el mundo real: El software debe usar el lenguaje del usuario, no terminología técnica innecesaria.
- Control y libertad del usuario: Los usuarios deben poder deshacer acciones fácilmente.
- Consistencia y estándares: Las funciones similares deben comportarse de la misma forma.
- Prevención de errores: Mejor prevenir errores que corregirlos después.
- Reconocimiento antes que memorización: Reducir la carga de memoria del usuario.
- Flexibilidad y eficiencia: Atajos para usuarios expertos sin entorpecer a los novatos.
- Estética y diseño minimalista: El diseño no debe contener información irrelevante.
- Ayuda y documentación: Proveer soporte cuando sea necesario.

Un software usable no solo funciona correctamente, sino que facilita al usuario alcanzar sus metas de forma simple y placentera.

Ejemplo: Una aplicación móvil de citas médicas que permite agendar una consulta en tres pasos simples, con mensajes claros y navegación intuitiva, presenta alta usabilidad.

1.6.2 Trabajo en equipo en el desarrollo de software

El software moderno rara vez es construido por una sola persona. Involucra equipos multidisciplinarios que integran programadores, diseñadores, analistas de requerimientos, testers y gerentes de proyecto.

Importancia del trabajo en equipo:

- Colaboración efectiva: Compartir información y esfuerzos de manera fluida.
- Comunicación clara: Evitar malentendidos técnicos o de requisitos.
- Coordinación de tareas: Asignar y sincronizar actividades en el tiempo.
- Gestión de conflictos: Resolver desacuerdos de manera constructiva.
- Confianza mutua: Fundamentar la relación de equipo en respeto y responsabilidad compartida.
- Factores críticos para el éxito del equipo:
- Liderazgo técnico y humano: Un líder efectivo guía tanto en lo técnico como en la cohesión del equipo.
- Roles claramente definidos: Cada miembro debe conocer sus responsabilidades específicas.
- Metas comunes: El equipo debe compartir una visión y un objetivo claros.
- Capacitación y mentoring: Favorecer el crecimiento técnico y profesional continuo.

McConnell (1996) enfatiza que: “La productividad de un proyecto de software depende más del talento, motivación y colaboración del equipo que de cualquier herramienta o proceso.”

Resumen del capítulo 1

En este primer capítulo se ha establecido una visión general de la Ingeniería de Software como disciplina estructurada, cuyo propósito es guiar el desarrollo de sistemas de software de manera sistemática, eficiente y orientada a la calidad. Se destacó que el software no es simplemente un conjunto de instrucciones, sino una solución lógica creada por personas y para personas, lo cual exige tanto conocimientos técnicos como sensibilidad hacia los aspectos humanos del proceso.

Se introdujeron los fundamentos de la Ingeniería de Software, sus capas conceptuales (calidad, proceso, métodos y herramientas), y se exploró el proceso de desarrollo desde una perspectiva metodológica. Se analizaron los principales modelos tradicionales de proceso —cascada, incremental y espiral—, junto con una mención preliminar a los enfoques ágiles que se abordarán en capítulos posteriores.

También se abordaron los costos del desarrollo, técnicas de estimación como COCOMO y Function Points, así como el impacto que tiene la corrección temprana de errores en la economía del proyecto. Se presentaron los atributos de calidad de un buen software según diversos autores y normas, incluyendo funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad.

Finalmente, se discutieron los retos fundamentales que enfrenta la disciplina, como la volatilidad de los requisitos, la complejidad creciente de los sistemas y la presión por calidad y tiempos de entrega. En este contexto, se resaltó la importancia de los aspectos humanos, especialmente la usabilidad del producto final y el trabajo colaborativo en equipos de desarrollo multidisciplinarios.

Este capítulo sienta las bases conceptuales para los contenidos que se abordarán en los siguientes capítulos, proporcionando al lector una comprensión global del rol de la Ingeniería de Software en la construcción de soluciones tecnológicas eficientes, sostenibles y centradas en el usuario.

Preguntas de revisión

1. Explique el concepto de Ingeniería de Software y argumente: ¿por qué se considera una disciplina de carácter tanto técnico como humano?.
2. Analice las capas que conforman la Ingeniería de Software según Pressman y Maxim. ¿Cómo se relacionan entre sí y qué papel juega la calidad en este modelo?
3. Compare al menos dos modelos tradicionales de proceso de software (como cascada, incremental o espiral). ¿En qué contextos es más conveniente aplicar cada uno?
4. Describa los factores que influyen en el costo del desarrollo de software y explique cómo el uso de modelos como COCOMO puede ayudar a una mejor planificación.
5. Mencione y explique tres atributos esenciales de un software de calidad. ¿Cómo impacta cada uno en la experiencia del usuario final?
6. Reflexione sobre los retos fundamentales que enfrenta actualmente la Ingeniería de Software. ¿Cuál considera usted que es el más crítico y por qué?
7. ¿Por qué es importante considerar la usabilidad y el trabajo en equipo como parte del proceso de desarrollo de software? Apoye su respuesta con ejemplos.

Autoevaluación

Parte A: Verdadero o Falso

Indique si las siguientes afirmaciones son Verdaderas (V) o Falsas (F).

1. El modelo en espiral de Boehm prioriza la entrega rápida de producto sin considerar la gestión de riesgos. ( )
2. El software embebido es un tipo de software que funciona de manera autónoma dentro de dispositivos físicos. ( )

3. Según Pressman y Maxim, la calidad en el software debe evaluarse únicamente en la fase de pruebas. ( )
4. La mantenibilidad es la facilidad con la cual un software puede ser transferido o ejecutado a un nuevo entorno operativo. ( )
5. La usabilidad es un atributo que influye directamente en la satisfacción del usuario final. ( )

Parte B: Selección Múltiple

Seleccione la opción correcta.

1. ¿Cuál de los siguientes factores NO influye directamente en el costo de un proyecto de software?
    a) Complejidad de los requisitos
    b) Número de usuarios finales
    c) Experiencia del equipo
    d) Tecnología utilizada
2. En el modelo incremental de desarrollo de software:
    a) Todo el sistema se entrega al final en una sola versión.
    b) Las funcionalidades se implementan y entregan de manera progresiva.
    c) No se realiza retroalimentación hasta finalizar el proyecto.
    d) Solo se modela la arquitectura inicial sin codificación.
3. ¿Cuál de los siguientes atributos describe mejor la portabilidad del software?
    a) Capacidad de soportar múltiples fallos.
    b) Facilidad de trasladarlo entre diferentes plataformas.
    c) Optimización del uso de recursos de hardware.
    d) Minimización del número de errores humanos.

Parte C: Relacione las columnas

Relacione cada concepto de la columna A con su definición correcta en la columna B.

Columna A	Columna B

a) Funcionalidad	( ) Capacidad de adaptarse a distintos entornos de operación.
b) Mantenibilidad	( ) Cumplimiento correcto de los requisitos establecidos.
c) Usabilidad	( ) Facilidad para ser aprendido y utilizado eficazmente por los usuarios.
d) Portabilidad	( ) Capacidad para corregir, mejorar o adaptar el sistema a cambios futuros.
e) Confiabilidad	( ) Habilidad del software para funcionar correctamente bajo condiciones específicas durante un tiempo determinado.

Ejercicio de aplicación final

"Entre entregas y errores: ¿qué salió mal con el sistema de matrículas?"

Contexto del caso:

La Universidad decidió modernizar sus procesos académicos implementando un sistema de matrículas en línea. El objetivo era mejorar la experiencia del estudiante, reducir tiempos en secretaría y optimizar la gestión de cupos. El proyecto fue adjudicado a una empresa local de desarrollo de software bajo un contrato que priorizaba el cumplimiento de plazos y costos, pero sin establecer cláusulas específicas sobre atributos de calidad ni mecanismos de soporte post-implementación.

El sistema fue entregado puntualmente, respetando el presupuesto asignado y superando las pruebas técnicas de

funcionamiento básico. Sin embargo, durante la primera semana del semestre se evidenciaron varios problemas:

- Muchos estudiantes manifestaron que la navegación era confusa y que los menús no seguían una lógica intuitiva.
- Al perderse la conexión a internet, los formularios no guardaban los datos, generando frustración y retrasos en el registro.
- Los técnicos de la universidad reportaron que modificar un módulo (como el de asignación de cupos) afectaba involuntariamente a otros componentes del sistema.
- La aplicación funcionaba únicamente en una versión específica de un navegador, dejando fuera a usuarios de otros entornos.
- No existían manuales de usuario claros ni un equipo de asistencia para resolver dudas o incidentes.

Estos problemas generaron malestar en estudiantes y docentes, además de sobrecargar al personal de TI de la universidad, que no había sido capacitado para dar soporte. La dirección académica cuestionó entonces si realmente el proyecto había sido exitoso, ya que se cumplió con el cronograma y el presupuesto, pero no con las expectativas de calidad y satisfacción de los usuarios.

Actividades para desarrollar:

1. Identifique al menos 4 atributos de calidad del software que no se cumplieron en este caso. Justifique su elección con base en el contenido del capítulo.
2. ¿Qué aspectos humanos del desarrollo fueron posiblemente ignorados durante el proyecto?
3. ¿Qué modelo de proceso tradicional (cascada, incremental, espiral) crees que fue aplicado? ¿Por qué?
4. Si hubieras sido parte del equipo, ¿qué acciones hubieras propuesto para evitar estos problemas?

Objetivos del ejercicio:

- Evaluar la capacidad del estudiante para aplicar conceptos clave del capítulo a una situación realista.
- Desarrollar habilidades de análisis crítico y resolución de problemas.
- Fomentar la discusión y reflexión sobre las decisiones que afectan la calidad del software.

Capítulo 2: Modelos ágiles en ingeniería de software

Introducción del capítulo

Durante décadas, el desarrollo de software estuvo dominado por modelos tradicionales que proponían procesos lineales, altamente estructurados y rígidos. Sin embargo, a medida que los entornos de negocios y las necesidades de los usuarios comenzaron a cambiar de manera vertiginosa, surgió la necesidad de adoptar enfoques más flexibles, adaptativos y centrados en la entrega rápida de valor. De esta necesidad emergieron los Modelos Ágiles.

Los modelos ágiles representan un cambio profundo de paradigma: ponen el énfasis en las personas, la colaboración, la adaptabilidad y la entrega temprana y continua de productos funcionales. Esta forma de trabajar no solo ha transformado el desarrollo de software, sino también la gestión de proyectos y la dinámica organizacional en empresas de todo el mundo.

Cita destacada:

"La agilidad no consiste en moverse más rápido; consiste en adaptarse mejor." — Jim Highsmith (coautor del Manifiesto Ágil)

En este capítulo, exploraremos los principios fundamentales del desarrollo ágil, los procesos característicos de los principales modelos ágiles, y analizaremos el costo del cambio en contextos ágiles frente a los métodos tradicionales. Además, estudiaremos cómo y cuándo resulta conveniente aplicar metodologías ágiles en la industria real, complementando el aprendizaje con casos de uso reales en áreas como salud, finanzas y comercio electrónico.

Adoptar una mentalidad ágil no significa rechazar la planificación o la calidad, sino aprender a equilibrarlas de manera inteligente frente a los cambios inevitables. Por ello, el estudio de los modelos ágiles no solo es una tendencia moderna, sino una competencia esencial para cualquier profesional de la Ingeniería de Software del siglo XXI.

Objetivos de aprendizaje

Al finalizar el estudio de este capítulo, el lector será capaz de:

- Comprender los principios fundamentales que sustentan el enfoque ágil en el desarrollo de software.
- Distinguir los modelos ágiles de los modelos tradicionales, identificando sus ventajas y limitaciones.
- Analizar el concepto del costo del cambio y cómo se gestiona en contextos ágiles.
- Describir los procesos característicos de metodologías ágiles como Scrum, XP y Kanban.
- Evaluar la aplicabilidad de los modelos ágiles en distintos sectores industriales.
- Interpretar casos de uso reales en los que los enfoques ágiles hayan sido aplicados exitosamente.

Esquema del capítulo

Preguntas guía

- ¿Qué motivó el surgimiento de los modelos ágiles en el desarrollo de software?
- ¿En qué se diferencia la gestión del cambio en métodos ágiles respecto a métodos tradicionales?
- ¿Qué principios del Manifiesto Ágil transformaron la forma en que se conciben los equipos y productos?
- ¿Qué metodologías ágiles existen y cuáles son sus elementos clave?
- ¿En qué contextos resulta más adecuado aplicar enfoques ágiles, y en cuáles no?

2.1 Introducción a los modelos de desarrollo ágil

Contexto histórico y necesidad del cambio

Durante gran parte del siglo XX, los proyectos de software se gestionaron bajo paradigmas estructurados y lineales, como el modelo en cascada. Estos enfoques, aunque adecuados en proyectos con requisitos bien definidos y entornos estables, comenzaron a mostrar severas limitaciones ante el entorno volátil, cambiante y competitivo de finales de los años noventa.

En ese período, muchas organizaciones comenzaron a experimentar:

- Cambios constantes de requisitos por parte de los clientes.
- Altas tasas de fracaso en proyectos largos y costosos.
- Dificultades para entregar valor en tiempo útil.
- Frustración por parte de los usuarios finales y equipos técnicos.

Según el informe del Standish Group (Chaos Report), más del 50% de los proyectos de software de esa época no cumplían con el alcance, presupuesto o cronograma original, y muchos eran cancelados antes de completarse.

Uno de los ejemplos más conocidos de fracaso en el uso de modelos tradicionales es el proyecto Virtual Case File (VCF) del FBI, desarrollado a inicios de los años 2000.

¿Qué era el VCF?

El VCF fue un sistema diseñado para reemplazar el antiguo sistema de archivos en papel del FBI, con el objetivo de digitalizar y modernizar el manejo de casos y expedientes criminales.

¿Qué ocurrió?

- El proyecto se desarrolló utilizando un enfoque en cascada, con especificaciones rígidas, documentación exhaustiva y poca interacción directa con los usuarios reales.

- No se entregaron versiones funcionales intermedias.
- Los requisitos fueron definidos inicialmente y se mantuvieron estáticos, incluso cuando las necesidades operativas cambiaron tras el 11-S.
- Cuando finalmente se presentó una versión del sistema, los agentes del FBI la consideraron inusable.

Resultados**:**

- Se invirtieron más de 170 millones de dólares en un sistema que nunca fue utilizado.
- El proyecto fue cancelado oficialmente en 2005.
- El FBI finalmente optó por un enfoque más ágil y colaborativo llamado Sentinel, que sí logró implementarse años después.

Cita destacada (OIG FBI Report, 2005): "El enfoque rígido de desarrollo, la falta de entregas iterativas y la desconexión con el usuario final fueron factores clave en el fracaso del VCF."

Frente a esta realidad, un grupo de 17 expertos en desarrollo de software se reunió en 2001 en Snowbird, Utah (EE. UU.), y redactó lo que se conoce como el Manifiesto Ágil.

El Manifiesto Ágil: punto de inflexión

El Manifiesto Ágil fue creado en febrero de 2001, cuando 17 expertos en desarrollo de software se reunieron en la estación de esquí de Snowbird, en Utah (EE. UU.), con el propósito de encontrar una alternativa a los métodos de desarrollo pesado que estaban generando frustración en la industria.

Estos expertos representaban diversas metodologías ligeras ya existentes en ese momento, como Extreme Programming (XP), Scrum, Crystal, Adaptive Software Development y Feature-Driven Development. De ese encuentro surgió el documento conocido como el Agile Manifesto, que revolucionó el desarrollo de software moderno.

Los 17 firmantes del Manifiesto Ágil fueron:

1. Kent Beck – Creador de Extreme Programming
2. Mike Beedle – Coautor de Scrum
3. Arie van Bennekum – Promotor de DSDM

4. Alistair Cockburn – Creador de Crystal
5. Ward Cunningham – Cocreador de Wiki y patrones de diseño
6. Martin Fowler – Autor y experto en arquitectura de software
7. James Grenning – Promotor del desarrollo ágil embebido
8. Jim Highsmith – Desarrollador de Adaptive Software Development
9. Andrew Hunt – Coautor de *The Pragmatic Programmer*
10. Ron Jeffries – Miembro del equipo original de XP
11. Jon Kern – Experto en diseño y arquitectura de software
12. Brian Marick – Especialista en pruebas ágiles
13. Robert C. Martin (Uncle Bob) – Promotor del desarrollo ágil y del "clean code"
14. Steve Mellor – Conocido por sus aportes al modelado orientado a objetos
15. Ken Schwaber – Cocreador de Scrum
16. Jeff Sutherland – Cocreador de Scrum
17. Dave Thomas – Coautor de *The Pragmatic Programmer*

Dato relevante**:** Estos profesionales venían de escuelas distintas, pero compartían una visión común: hacer que el desarrollo de software fuera más humano, flexible y efectivo.

El Manifiesto Ágil no es una metodología, sino una declaración de valores y principios que redefinen la manera en que se concibe, planifica, construye y entrega software.

Los 4 valores fundamentales del Manifiesto Ágil son:

1. Individuos e interacciones sobre procesos y herramientas
2. Software funcionando sobre documentación extensiva
3. Colaboración con el cliente sobre negociación de contratos
4. Respuesta ante el cambio sobre seguir un plan rígido

Manifiesto Ágil propone 12 principios que orientan el comportamiento y las prácticas de los equipos de desarrollo. Estos principios fueron formulados para abordar los

problemas recurrentes del desarrollo tradicional, promoviendo flexibilidad, colaboración y entrega continua de valor.

A continuación, se presenta un resumen interpretativo de cada uno:

1. Satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptar cambios en los requisitos, incluso en etapas avanzadas del desarrollo.
3. Entregar software funcional frecuentemente, desde semanas hasta meses, con preferencia por intervalos cortos.
4. Colaboración constante entre el equipo de desarrollo y los usuarios o clientes.
5. Motivar a los individuos y confiar en ellos para realizar el trabajo.
6. Comunicación cara a cara como el método más eficiente para transmitir información.
7. Software funcionando como principal medida de progreso.
8. Desarrollo sostenible, manteniendo un ritmo constante e indefinido.
9. Atención continua a la excelencia técnica y buen diseño.
10. Simplicidad, entendida como el arte de maximizar el trabajo no realizado.
11. Equipos autoorganizados que generan mejores diseños y arquitecturas.
12. Reflexión periódica sobre cómo ser más efectivos, ajustando el comportamiento en consecuencia.

Estos principios guían tanto la acción como la cultura del equipo, alineando la práctica diaria con una mentalidad de mejora continua.

"Los métodos ágiles cambiaron la forma en que las organizaciones piensan sobre el software. Ya no se trata de una entrega final, sino de una entrega continua de valor." — Highsmith (2002)

¿Qué son los procesos ágiles?

En el contexto del desarrollo de software, un proceso ágil se refiere a una serie de prácticas organizadas que guían la forma en que un equipo construye, entrega y mejora un producto. Estos procesos no siguen un camino fijo y lineal, sino que están diseñados para ser iterativos, adaptativos y centrados en el valor entregado al cliente. A diferencia de los modelos tradicionales, donde se intenta predecir todo desde el inicio, los procesos ágiles aceptan que la incertidumbre es parte inherente al desarrollo y plantean mecanismos estructurados para aprender y ajustarse en el camino.

Un proceso ágil se caracteriza por estar dividido en ciclos cortos de trabajo (iteraciones o sprints), donde en cada uno se produce un incremento funcional del producto. Al final de cada iteración, se revisa lo entregado, se recibe retroalimentación y se planifica el siguiente paso. Este enfoque permite ajustar continuamente los requisitos, prioridades y soluciones técnicas en función de la evolución del entorno y las necesidades del usuario.

Lo esencial de un proceso ágil no es simplemente ser más rápido, sino ser más inteligente en la forma de construir software: validar lo que funciona lo antes posible, eliminar desperdicios, colaborar estrechamente entre miembros del equipo y con los clientes, y fomentar la mejora continua.

Los procesos ágiles se basan en los valores y principios del Manifiesto Ágil, pero cada organización los implementa de manera distinta según su contexto. Por ejemplo, una empresa puede seguir un proceso basado en Scrum, mientras otra adopta Kanban o combina prácticas de varias metodologías. Lo importante no es el nombre del proceso, sino que esté

orientado a entregar valor temprano, sostener calidad y adaptarse al cambio.

Un ejemplo notable de implementación exitosa de procesos ágiles se encuentra en la empresa ING Bank en los Países Bajos. A partir de 2015, ING inició una transformación organizacional a gran escala inspirada en los modelos de Spotify y Scrum. Reestructuraron sus equipos en *squads* multidisciplinarios, eliminaron jerarquías innecesarias y adoptaron procesos ágiles para gestionar sus servicios digitales. Como resultado, lograron reducir el tiempo de entrega de nuevas funcionalidades de meses a semanas, mejorar la colaboración interdepartamental y aumentar la satisfacción de los usuarios finales. Este caso muestra cómo los procesos ágiles pueden aplicarse más allá de la industria tecnológica, en sectores altamente regulados como el financiero.

Ser ágil implica:

- Priorizar entregas incrementales y funcionales.
- Obtener retroalimentación temprana y continua.
- Favorecer la colaboración directa entre desarrolladores y clientes.
- Adaptarse a nuevos requerimientos con flexibilidad.

Tabla 3

Comparación con modelos tradicionales

Enfoque Tradicional	**Enfoque Ágil**
Planificación detallada inicial	Planificación continua e iterativa
Entrega única al final	Entregas frecuentes e incrementales
Contrato como base de gestión	Colaboración directa con el cliente
Roles jerárquicos	Equipos autoorganizados

Documentación exhaustiva	Documentación ligera y útil

Nota. Elaboración propia

Ventajas del enfoque ágil

- Reducción del tiempo de entrega.
- Mejor alineación con el cliente.
- Aumento de la motivación y compromiso del equipo.
- Flexibilidad para adaptarse al cambio.
- Mayor control de calidad a través de entregas continuas.

Limitaciones y consideraciones

Aunque los modelos ágiles ofrecen muchas ventajas, no son universales. No son ideales en:

- Proyectos donde los requisitos son completamente estables desde el inicio.
- Entornos altamente regulados que exigen trazabilidad estricta.
- Equipos sin experiencia en autogestión ni disciplina técnica.

Sommerville (2016) señala que el enfoque ágil requiere una madurez organizacional que no todos los contextos poseen.

2.2 Procesos y metodologías ágiles más representativas

El enfoque ágil ha dado lugar a múltiples marcos de trabajo que adoptan los valores y principios del Manifiesto. A continuación, se presentan las metodologías ágiles más influyentes:

Marco de trabajo, metodología y método: ¿es lo mismo?

En el estudio de los enfoques ágiles es común encontrar confusiones terminológicas entre conceptos como *marco de trabajo*, *metodología* y *método*. Aunque a veces se usan como sinónimos en el lenguaje informal, tienen significados distintos desde el punto de vista técnico.

Un marco de trabajo (framework) proporciona una estructura general, con roles, eventos y artefactos definidos, pero permite flexibilidad sobre cómo aplicar cada práctica. No indica con exactitud cómo desarrollar el software. Scrum es un ejemplo de marco de trabajo.

Una metodología es más detallada y prescriptiva: incluye fases, prácticas, principios y herramientas que deben seguirse paso a paso. Extreme Programming (XP) es una metodología, ya que define prácticas específicas como programación en parejas, desarrollo guiado por pruebas (TDD) e integración continua.

Un método es aún más ligero: se enfoca en la gestión del flujo de trabajo, sin imponer estructuras organizativas o técnicas específicas. Kanban es un método, ya que se basa en visualización, límites de trabajo en curso (WIP) y mejora continua del flujo.

Tabla 4

Comparación de metodologías ágiles

Enfoque	**Tipo**	**Nivel de prescripción**	**Ejemplo de uso**
Scrum	Marco de trabajo	Medio	Gestión de proyectos iterativos
XP	Metodología	Alto	Desarrollo técnico con prácticas ágiles
Kanban	Método	Bajo	Flujo continuo en mantenimiento

Nota. Elaboración propia

¿Cómo llamarlos en conjunto?

Para referirse a Scrum, XP y Kanban en conjunto, lo más adecuado es utilizar expresiones como:

- "Enfoques Ágiles de Desarrollo"
- "Modelos Ágiles"
- "Prácticas de Desarrollo Ágil" (en contextos técnicos)

Esto evita clasificaciones incorrectas y respeta las diferencias entre cada enfoque.

2.2.1 Scrum: Marco ágil para el desarrollo de software

Fundamentos de Scrum

Scrum es un marco de trabajo ágil para la gestión de proyectos complejos, especialmente útil en desarrollo de software. Fue formulado por Ken Schwaber y Jeff Sutherland en la década de 1990 y formalizado en la guía *Scrum Guide*.

En ese artículo, se destacaba que los proyectos exitosos no seguían procesos secuenciales estrictos, sino que trabajaban de forma iterativa, adaptativa y en equipo: justo como en un scrum.

Nota Cultural: ¿De dónde viene el nombre *Scrum*?

El término *Scrum* no fue inventado originalmente para el desarrollo de software. Proviene del rugby, donde hace referencia a una formación en la que los jugadores se alinean hombro con hombro para empujar juntos por el control del balón. Esta metáfora fue tomada por los pioneros de la agilidad para representar el trabajo colaborativo, coordinado y sin jerarquías rígidas que caracteriza a los equipos ágiles.

La idea fue introducida por primera vez en el ámbito de la gestión de software en un artículo publicado por Takeuchi y Nonaka en 1986, titulado *"The New Product Development Game"* (Harvard Business Review), donde argumentaban que los equipos de alto rendimiento en innovación se asemejaban más a un "scrum" de rugby que a una línea de ensamblaje.

Este concepto inspiró a Ken Schwaber y Jeff Sutherland, quienes tomaron el término para nombrar el marco de trabajo que hoy conocemos como Scrum.

Scrum no es una metodología en el sentido tradicional; no prescribe técnicas específicas de desarrollo, sino que establece roles, eventos y artefactos que deben ser adaptados por los equipos para alcanzar objetivos de producto de forma iterativa e incremental.

Roles en Scrum

Scrum define tres roles principales, claramente delimitados:

1. Product Owner (Propietario del Producto)

- Representa los intereses del cliente o usuario.
- Define y prioriza las características del producto (historias de usuario).
- Es responsable del Product Backlog y de maximizar el valor del trabajo del equipo.

2. Scrum Master

- Actúa como facilitador del equipo.
- Elimina impedimentos, asegura que se sigan los principios de Scrum.
- No es un jefe de proyecto, sino un líder servicial.

3. Development Team (Equipo de Desarrollo)

- Grupo multifuncional (programadores, diseñadores, testers, etc.).
- Se autoorganiza para entregar un incremento funcional en cada Sprint.
- Tienen autonomía para decidir cómo realizar el trabajo.

Eventos (Ceremonias Scrum)

Scrum establece eventos regulares y definidos para mantener la transparencia, inspección y adaptación.

1. Sprint

- Ciclo de desarrollo fijo (generalmente 2 a 4 semanas).
- Al finalizar, se entrega un incremento funcional del producto.
- Durante el Sprint no se deben cambiar los objetivos.

2. Sprint Planning (Planificación del Sprint)

- Se define qué se desarrollará en el Sprint y cómo se realizará.
- Participan el Product Owner y el equipo de desarrollo.
- Resultado: Sprint Goal y el Sprint Backlog.

3. Daily Scrum (Reunión Diaria)

- Reunión de 15 minutos todos los días del Sprint.
- El equipo sincroniza actividades y discute obstáculos.
- Se responde a tres preguntas clave:
    - ¿Qué hice ayer?
    - ¿Qué haré hoy?
    - ¿Qué impedimentos tengo?

4. Sprint Review (Revision del Sprint)

- Se muestra el incremento funcional al Product Owner y stakeholders.
- Se recibe retroalimentación para refinar el Product Backlog.

5. Sprint Retrospective

- Se realiza al final de cada Sprint.
- El equipo reflexiona sobre lo que funcionó y lo que no.
- Se identifican acciones de mejora para el siguiente Sprint.

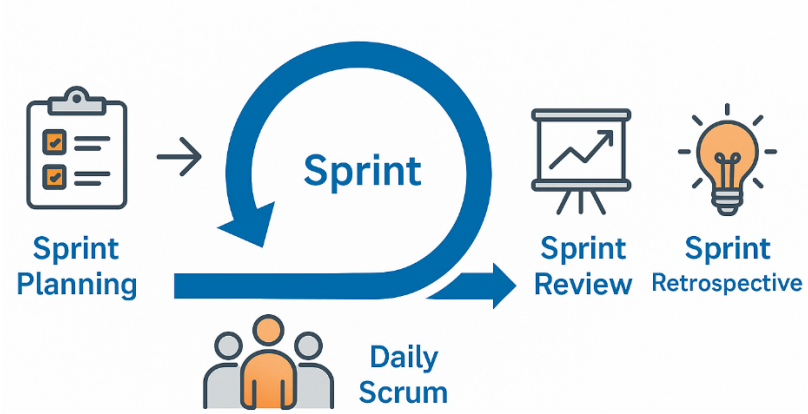

Figura 1
Secuencia de eventos durante un sprint
Nota. Elaboración propia

Artefactos de Scrum

Dentro del marco de Scrum, los artefactos representan los elementos tangibles que permiten visualizar el trabajo, mantener la transparencia del proceso y fomentar la inspección continua. Cada artefacto está diseñado para brindar información crítica sobre el producto en desarrollo, el progreso del equipo y las decisiones que deben tomarse. Scrum define tres artefactos principales: el Product Backlog, el Sprint Backlog y el Incremento

El Product Backlog es una lista ordenada y priorizada de todo lo que se necesita para el desarrollo del producto. Funciona como la única fuente autorizada de requisitos del sistema o funcionalidad que se está construyendo. Este artefacto está en constante evolución: a medida que se aprende más sobre el producto y el entorno, el Product Backlog se actualiza, se refina y se reordena. Su gestión es responsabilidad del Product Owner, quien se encarga de priorizar los elementos

en función del valor que aportan al cliente o usuario final. Cada ítem del Product Backlog se denomina "elemento de backlog" (backlog item), y puede representarse como historias de usuario, tareas técnicas, correcciones o mejoras2. Sprint Backlog

Por otro lado, el Sprint Backlog es un subconjunto del Product Backlog seleccionado por el equipo durante la planificación del Sprint. Contiene los elementos del producto que se comprometen a entregar durante el Sprint, así como el plan detallado para completarlos. Este artefacto proporciona visibilidad sobre el trabajo que el equipo ha decidido ejecutar y permite hacer seguimiento diario al progreso. El Sprint Backlog es dinámico: puede ajustarse durante el Sprint en función de los avances, siempre que no se altere el objetivo del Sprint acordado.

Finalmente, el Incremento es el resultado del trabajo completado durante un Sprint, combinado con los incrementos de todos los Sprints anteriores. Debe representar una versión del producto que sea funcional, integrada y potencialmente entregable. Para que un ítem se considere parte del Incremento, debe cumplir con los criterios de calidad definidos por el equipo, conocidos como la "Definition of Done" (definición de terminado). Este artefacto es esencial para medir el progreso real y tangible del equipo en cada iteración.

En conjunto, estos artefactos permiten que Scrum se mantenga como un marco transparente y basado en evidencia, donde las decisiones se toman sobre la base de información visible, actualizada y compartida por todos los actores del proyecto

Ventajas de Scrum

- Entregas tempranas y frecuentes de valor.
- Mejora continua del proceso.
- Alta visibilidad del progreso.
- Fomenta la colaboración y responsabilidad compartida.

- Adaptabilidad ante el cambio.

Limitaciones

- Requiere madurez organizacional y disciplina.
- Puede fallar si los roles no se respetan o no hay compromiso.
- No es adecuado para proyectos donde los requisitos son completamente fijos desde el inicio.

Aplicabilidad

Scrum es ampliamente utilizado en:

- Desarrollo de productos de software
- Startups tecnológicas
- Equipos distribuidos
- Proyectos que requieren alta interacción con el cliente
- Contextos donde la retroalimentación continua es valiosa

Empresas como Google, Spotify, IBM, Microsoft y muchas otras han adoptado Scrum (o variantes) como parte de su cultura de desarrollo.

Spotify y su adaptación del modelo Scrum

Una de las implementaciones más influyentes de Scrum, aunque adaptada a gran escala, es la realizada por la empresa Spotify, el conocido servicio de streaming musical. Spotify adoptó los principios de Scrum en sus primeras etapas de crecimiento para coordinar el trabajo de múltiples equipos de desarrollo, distribuidos geográficamente y centrados en la entrega continua de funcionalidades innovadoras para millones de usuarios.

A medida que la empresa creció, su equipo de ingeniería transformó la implementación tradicional de Scrum en un modelo organizacional más flexible, conocido como el modelo de escuadrones (squads), tribus y capítulos. Cada *squad* funciona como un equipo Scrum autónomo, responsable de una funcionalidad específica del producto, y sigue los principios de iteración corta, retrospectivas,

reuniones diarias y definición clara de roles, aunque adaptados a su contexto.

Este enfoque les permitió mantener una cultura ágil incluso con miles de empleados, minimizando la burocracia y favoreciendo la innovación continua. Spotify no siguió al pie de la letra la Scrum Guide, pero sí se apoyó en sus principios para fomentar la autoorganización, la transparencia, la entrega incremental de valor y la mejora continua.

Los resultados fueron notables: la empresa logró iterar rápidamente, lanzar funciones experimentales y escalar sus operaciones sin perder agilidad. Spotify se convirtió así en un referente global de cómo los principios de Scrum pueden inspirar estructuras organizativas ágiles, incluso en escenarios de crecimiento masivo.

Este caso demuestra que Scrum, más que un conjunto de reglas fijas es un marco que puede evolucionar y adaptarse para responder a contextos diversos, siempre que se conserven sus principios fundamentales de colaboración, iteración y entrega constante de valor.

2.2.2 Extreme Programming (XP): Desarrollo técnico con agilidad extrema

Extreme Programming (XP) es una metodología ágil que pone el énfasis en las prácticas de desarrollo técnico para mejorar tanto la calidad del software como la capacidad de adaptarse a cambios frecuentes en los requisitos. Fue desarrollada por Kent Beck a mediados de los años noventa mientras lideraba un proyecto en Chrysler, conocido como C3 (Chrysler Comprehensive Compensation). En ese contexto, los métodos tradicionales no estaban dando resultados satisfactorios, por lo que Beck propuso un conjunto de prácticas intensivas de desarrollo centradas en la simplicidad, la retroalimentación continua y la responsabilidad colectiva del código.

A diferencia de Scrum, que se enfoca más en la gestión del proceso, XP profundiza en el cómo se debe escribir código de alta calidad dentro de un marco ágil. Esta metodología

promueve la entrega temprana y frecuente de pequeñas versiones del software, apoyada en prácticas técnicas estrictas que buscan minimizar errores, facilitar el mantenimiento y garantizar que el producto evolucione sin degradarse.

Uno de los pilares de XP es la programación en parejas (pair programming), en la cual dos desarrolladores trabajan juntos en una misma estación de trabajo: uno escribe el código mientras el otro revisa en tiempo real, fomentando el control de calidad desde el origen. Otra práctica central es el desarrollo guiado por pruebas (TDD, por sus siglas en inglés), que implica escribir primero las pruebas automatizadas antes del código funcional, asegurando que cada funcionalidad cumpla exactamente con los requerimientos esperados.

La integración continua es también una práctica clave: el código se integra varias veces al día en un repositorio central, permitiendo detectar conflictos o errores a tiempo. Además, XP fomenta la propiedad colectiva del código, el diseño simple, la comunicación constante con el cliente y un ritmo de desarrollo sostenible.

Entre sus principios fundamentales se destacan:

- La retroalimentación constante, tanto del cliente como del propio sistema mediante pruebas automatizadas.
- La simplicidad del diseño, desarrollando solo lo necesario en cada iteración.
- El coraje para cambiar lo que no funciona, y para proponer mejoras incluso cuando se avanza en una dirección distinta.
- El respeto mutuo dentro del equipo, reconociendo que todos comparten responsabilidad sobre el producto.

XP es especialmente eficaz en entornos donde los requisitos cambian con frecuencia, los equipos son pequeños o medianos, y se cuenta con un alto nivel de disciplina técnica. No obstante, su aplicación puede ser más desafiante en

organizaciones con estructuras rígidas o con equipos que no están familiarizados con buenas prácticas de desarrollo.

Un caso real que evidencia la efectividad de XP fue el proyecto C3 de Chrysler, donde su implementación ayudó a mantener una tasa baja de defectos y permitió realizar adaptaciones constantes en un entorno con requerimientos cambiantes. Aunque el proyecto eventualmente fue cancelado por razones organizacionales ajenas al equipo de desarrollo, la experiencia fue fundamental para validar el enfoque de XP y propulsar la adopción de metodologías ágiles en la industria.

Extreme Programming es una metodología que lleva al extremo las buenas prácticas de desarrollo con el fin de lograr un software de alta calidad, adaptable, probado y mantenible. Su foco está en la excelencia técnica como medio para alcanzar agilidad real y sostenible en el tiempo.

A diferencia de modelos tradicionales (como el cascada) que definen fases secuenciales claras (análisis, diseño, codificación, pruebas, mantenimiento), XP no está organizada en fases rígidas, sino que se basa en iteraciones breves (entre 1 y 2 semanas), cada una de las cuales incluye todas las actividades del ciclo de vida del software.

Martin Fowler y Kent Beck han sugerido un marco general de implementación por niveles o "ciclos de maduración" de XP, en el que un equipo puede adoptar inicialmente un conjunto de prácticas básicas e ir evolucionando hacia un dominio más disciplinado.

¿Cómo se desarrolla un proyecto con XP?

Un proyecto basado en XP normalmente sigue un flujo como este:

1. Exploración. El cliente define las funcionalidades deseadas en forma de historias de usuario. El equipo evalúa la viabilidad técnica y prioriza las historias según su valor.
2. Planeación de la Iteración (Iteration Planning). Se seleccionan las historias a implementar en la próxima

iteración. El equipo estima el esfuerzo y define las tareas técnicas necesarias.
3. Diseño simple. Antes de codificar, el equipo define un diseño lo más simple posible que permita cumplir los requisitos actuales. El diseño no debe anticiparse a funcionalidades futuras innecesarias.
4. Programación y pruebas. Se programa en parejas (pair programming) y se utilizan pruebas automatizadas desde el inicio. Se aplica TDD (Test-Driven Development): primero se escribe la prueba, luego el código que la hace pasar.
5. Integración continua. Cada vez que un fragmento de código está listo, se integra con el resto del sistema. Esto ocurre varias veces al día.
6. Entrega frecuente. Al final de cada iteración, se presenta al cliente una versión funcional del software que incorpora las historias de usuario acordadas.
7. Retroalimentación y adaptación. El cliente evalúa lo entregado y puede ajustar prioridades o introducir cambios. El equipo también reflexiona sobre su rendimiento.

Este ciclo se repite durante toda la vida del proyecto.

Ventajas de Extreme Programming (XP)

1. Alta calidad técnica del software. Gracias a prácticas como TDD, integración continua y refactorización constante, se minimizan los errores y se facilita el mantenimiento.
2. Entrega continua de valor. Cada iteración produce un incremento funcional que el cliente puede usar, evaluar y refinar.
3. Excelente gestión del cambio. XP está diseñado para adaptarse a cambios de requisitos en cualquier momento, incluso en etapas avanzadas.
4. Colaboración intensa con el cliente. El cliente participa activamente en el desarrollo, proporcionando retroalimentación constante.

5. Equipo motivado y responsable. La programación en parejas y la propiedad colectiva del código promueven la cohesión y el aprendizaje entre los miembros del equipo.

Limitaciones de Extreme Programming (XP)

1. Alta exigencia técnica. XP requiere un equipo disciplinado, con sólidos conocimientos en pruebas automatizadas, diseño simple, refactorización y trabajo colaborativo. No es adecuado para equipos novatos.
2. Difícil de escalar. XP funciona mejor en equipos pequeños (de 5 a 10 personas). Escalar XP a múltiples equipos o grandes organizaciones requiere una arquitectura complementaria no contemplada por la metodología original.
3. Dependencia del cliente. XP exige que el cliente esté disponible permanentemente para dar retroalimentación. Si esto no se cumple, el proceso se debilita.
4. Puede ser percibido como caótico sin entendimiento profundo. Si se aplican solo algunas prácticas aisladas (por ejemplo, pair programming sin pruebas), puede generar frustración y resultados poco sostenibles.

En conclusión, XP es una metodología robusta para el desarrollo ágil cuando se busca calidad técnica, rapidez de respuesta al cambio y colaboración cercana con el cliente. Su estructura iterativa y práctica permite responder a entornos dinámicos, pero exige disciplina, experiencia y compromiso del equipo. No tiene fases formales, pero sí un flujo de trabajo repetitivo donde todas las prácticas técnicas y de gestión se aplican en cada ciclo, enfocándose en mejorar constantemente tanto el producto como la forma en que se construye.

Caso de Éxito: Desarrollo de JUnit y el enfoque XP de Kent Beck

Uno de los casos más representativos del uso exitoso de Extreme Programming es la creación del marco de pruebas

JUnit, ampliamente utilizado en el desarrollo de software en Java. Este proyecto fue desarrollado por Kent Beck y Erich Gamma a finales de los años noventa como una herramienta para automatizar pruebas unitarias, coincidiendo con la consolidación del enfoque XP.

JUnit no solo se diseñó para facilitar la práctica del desarrollo guiado por pruebas (TDD), sino que fue desarrollado *usando* TDD y otras prácticas propias de XP desde el inicio. Kent Beck, su creador, aplicó las siguientes prácticas XP durante el desarrollo:

- Diseño simple: en lugar de construir un marco complejo desde el inicio, desarrollaron únicamente lo necesario para que las pruebas funcionaran, dejando que el diseño evolucionara naturalmente con el código.
- Desarrollo dirigido por pruebas (TDD): cada nueva funcionalidad fue precedida por una prueba que guiaba su implementación. Esto ayudó a crear una base sólida, extensible y fácilmente refactorizable.
- Integración continua: cada modificación se integraba inmediatamente en el repositorio principal, asegurando que el marco estuviera siempre funcional y estable.
- Programación en parejas: Beck y Gamma trabajaron juntos en sesiones intensas, revisando en tiempo real cada fragmento de código producido.

Gracias a estas prácticas, JUnit logró convertirse en una herramienta robusta, sencilla de usar, extensible y altamente confiable. Su influencia fue tan significativa que, actualmente, la mayoría de las herramientas de pruebas modernas se basan en su arquitectura y TDD se convirtió en una práctica estándar en muchas organizaciones que adoptaron metodologías ágiles.

Más allá del valor técnico del producto, JUnit se convirtió en un caso ejemplar de cómo XP permite construir software de alta calidad, desde pequeños proyectos hasta bibliotecas centrales de la industria.

2.2.3 Kanban: Flujo visual y mejora continua

Kanban es un método ágil de gestión del trabajo que se centra en la visualización del flujo de tareas, el control del trabajo en curso y la mejora continua del proceso. A diferencia de metodologías como XP o marcos como Scrum, Kanban no impone roles específicos ni iteraciones fijas, lo que lo convierte en una opción altamente flexible para equipos que necesitan optimizar sus flujos sin modificar de forma drástica su estructura organizacional.

El origen de Kanban se remonta al sistema de producción de Toyota, donde fue utilizado como una herramienta para gestionar inventarios y procesos de fabricación mediante señales visuales. Esta lógica fue adaptada al desarrollo de software por David J. Anderson en la década de 2000, quien lo formalizó como un método para gestionar sistemas de trabajo con enfoque en eficiencia, calidad y transparencia.

En Kanban, el trabajo se representa mediante tarjetas (tasks o ítems) que se mueven a través de columnas en un tablero, reflejando su estado actual. Por ejemplo, un tablero típico puede tener columnas como “Por hacer”, “En proceso” y “Terminado”. Esta visualización permite a todos los miembros del equipo y partes interesadas ver claramente el estado del proyecto en todo momento.

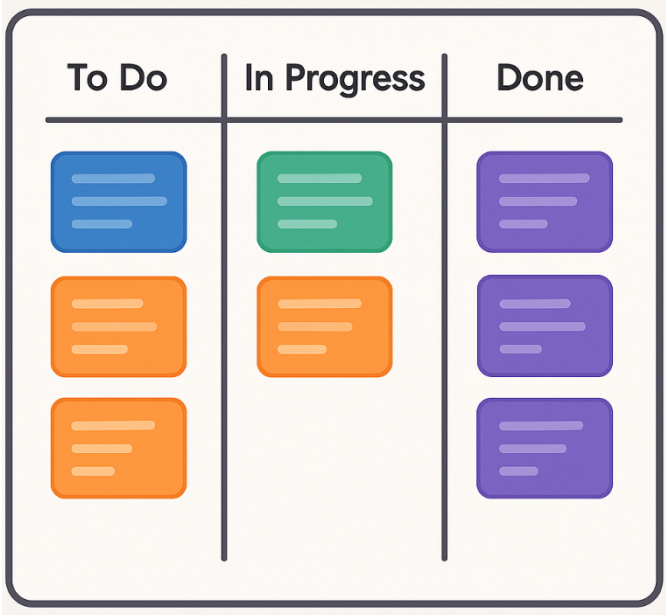

Figura 2
Tablero Kanban
Nota. Elaboración propia

Uno de los principios clave de Kanban es limitar el trabajo en curso. Esta práctica evita la multitarea excesiva, reduce los cuellos de botella y mejora el enfoque del equipo. Cuando se alcanza el límite en una columna, el equipo debe resolver los ítems actuales antes de comenzar nuevos, promoviendo la finalización en lugar de la iniciación constante.

Además, Kanban promueve la gestión evolutiva del proceso. No exige una transformación organizacional radical, sino que parte del proceso actual del equipo y lo mejora progresivamente a través de la observación, la retroalimentación y la toma de decisiones basada en datos.

Entre sus prácticas habituales también se encuentran la medición del tiempo de ciclo (lead time), el análisis de bloqueos, las reuniones de revisión del flujo (flow review) y la incorporación de políticas explícitas que definen cómo se trabaja en cada etapa del proceso.

Aunque Kanban no impone iteraciones fijas como los sprints de Scrum, los equipos pueden combinar ambos enfoques: por ejemplo, usando un tablero Kanban dentro de un marco Scrum para gestionar mejor las tareas de cada sprint.

Ventajas de Kanban

- Es fácil de adoptar y no requiere una reestructuración organizacional.
- Aporta visibilidad total del trabajo en curso.
- Mejora el tiempo de entrega al evitar sobrecarga de tareas.
- Favorece la entrega continua sin necesidad de iteraciones fijas.
- Promueve la mejora continua basada en datos y métricas reales.
- Limitaciones de Kanban
- No prescribe roles ni eventos, lo que puede dificultar su implementación si el equipo carece de autonomía o disciplina.
- Puede ser menos efectivo en proyectos que requieren una planificación rigurosa o entregables altamente estructurados.
- El éxito de Kanban depende de la madurez del equipo para autorregularse y revisar constantemente sus procesos.

Kanban en el equipo de IT de BBC Worldwide

Un caso ampliamente documentado del uso exitoso de Kanban se dio en el departamento de tecnología de BBC Worldwide, la unidad comercial de la cadena pública británica. A mediados de los años 2010, varios equipos internos comenzaron a enfrentar problemas de visibilidad, cuellos de botella en tareas críticas y entregas inestables.

En lugar de adoptar un enfoque ágil completo como Scrum, el equipo de desarrollo optó por introducir Kanban de forma

incremental**.** Se establecieron tableros físicos con límites de trabajo en curso, se identificaron los cuellos de botella y se comenzó a medir el tiempo de ciclo para cada tipo de tarea. También se establecieron reuniones breves para revisar el flujo de trabajo y ajustar el proceso según los datos observados.

En pocos meses, el equipo reportó mejoras significativas: se redujo el tiempo de entrega promedio en más del 30 %, aumentó la transparencia entre departamentos, y se detectaron patrones de trabajo ineficientes que antes pasaban desapercibidos. Lo más destacable es que estos logros se alcanzaron sin cambiar el equipo ni la estructura formal del departamento, demostrando que Kanban puede ser una poderosa herramienta de optimización sin necesidad de transformaciones drásticas

Kanban es un método ligero, visual y adaptable que permite gestionar el flujo de trabajo de manera efectiva, especialmente en entornos donde se requiere flexibilidad operativa, mejora y entrega continuas de valor. Si bien no proporciona una estructura completa como otras metodologías ágiles, su enfoque pragmático y evolutivo lo convierte en una excelente opción para organizaciones que buscan agilidad sin rigidez metodológica.

Tabla 5

Comparación entre Scrum, XP y Kanban

Aspecto	Scrum	XP (Extreme Programming)	Kanban
Estructura	Marco de trabajo	Metodología	Método de gestión de flujo
Iteraciones	Sprints fijos (2-4 semanas)	Iteraciones cortas (1-2 semanas)	No requiere iteraciones fijas

Enfoque principal	Gestión del proyecto y entregas incrementales	Calidad técnica y adaptabilidad	Optimización del flujo de trabajo
Roles definidos	Sí (Scrum Master, Product Owner, Equipo de desarrollo)	No formalizados como en Scrum	No definidos
Técnicas de desarrollo	No especificadas	Sí (TDD, integración continua, pair programming)	No prescritas, se pueden complementar
Gestión del cambio	Moderado, basado en revisión del backlog	Muy alto, permite cambios incluso tardíos	Basado en gestión visual y límites WIP
Tamaño de equipo ideal	5-9 personas	3-10 personas	Cualquier tamaño (flexible)
Adecuado para	Proyectos de productos con entregas frecuentes	Proyectos con alta incertidumbre técnica o cambios frecuentes	Mantenimiento, soporte continuo, mejora de procesos

Nota. Elaboración propia

En conclusión: ¿Cuándo elegir Scrum, XP o Kanban?

Cada enfoque ágil tiene fortalezas particulares que lo hacen más adecuado según el contexto del proyecto y las necesidades del equipo.

Scrum es ideal cuando se requiere una gestión estructurada con entregas frecuentes y se cuenta con un equipo que puede asumir roles definidos. Es especialmente útil en proyectos de desarrollo de productos donde la interacción con los usuarios es constante y se busca una mejora iterativa.

Extreme Programming (XP) resulta más apropiado cuando el proyecto demanda una alta calidad técnica, frecuentes cambios en los requisitos y un equipo altamente disciplinado. Su énfasis en prácticas como TDD y la integración continua lo convierten en una opción potente para contextos técnicamente complejos.

Kanban es la mejor opción para equipos que desean mejorar la eficiencia de su flujo de trabajo sin adoptar un marco rígido. Es especialmente eficaz en entornos de soporte, mantenimiento y operaciones, donde la continuidad y la adaptabilidad son más importantes que la planificación iterativa.

Elegir entre Scrum, XP o Kanban no implica adherirse exclusivamente a uno de ellos; en la práctica, muchas organizaciones combinan elementos de estos enfoques para construir un proceso ágil que se ajuste a sus objetivos, cultura y madurez técnica.

2.3 El costo del cambio en entornos ágiles

Una de las premisas fundamentales del desarrollo ágil es que el cambio no solo es inevitable, sino deseable. En contraste con los enfoques tradicionales, que tratan de evitar o minimizar los cambios una vez iniciado el desarrollo, las metodologías ágiles aceptan y se adaptan al cambio como parte natural del proceso creativo. Comprender cómo se gestiona el costo del cambio es clave para valorar la agilidad como estrategia.

En los modelos tradicionales, como el cascada o el espiral, el costo de introducir cambios suele aumentar drásticamente con el tiempo. Esta concepción fue representada gráficamente en lo que se conoce como la "curva del costo del cambio", en la que las modificaciones realizadas en fases

avanzadas del proyecto (como pruebas o mantenimiento) son mucho más costosas que si se hubieran hecho al inicio. El razonamiento detrás de este modelo está ligado al esfuerzo adicional que implica modificar requisitos, rediseñar componentes, reescribir código y realizar nuevas pruebas, una vez que todo ha sido definido y construido bajo un enfoque secuencial.

Sin embargo, esta visión fue desafiada por los creadores de XP y otros autores ágiles como Martin Fowler, quienes argumentaron que, si se aplican buenas prácticas técnicas (pruebas automatizadas, integración continua, diseño simple, refactorización), el costo del cambio no tiene por qué crecer de manera exponencial. En su lugar, puede mantenerse relativamente plano durante el ciclo de vida del proyecto.

En entornos ágiles, el cambio es gestionado mediante:

- Iteraciones cortas, que permiten revisar frecuentemente las decisiones tomadas.
- Feedback continuo, tanto del cliente como del propio sistema (pruebas automatizadas).
- Diseños evolutivos, que no intentan anticipar todos los escenarios desde el inicio, sino que se adaptan de manera orgánica al cambio.
- Entrega incremental, que reduce el riesgo acumulado al trabajar con versiones funcionales desde el principio.

Un ejemplo claro de la importancia de esta filosofía se evidenció en el rediseño de la interfaz de usuario de Gmail (Google). A lo largo de sus primeros años, el equipo de desarrollo realizó múltiples cambios en el diseño y las funcionalidades clave en función del comportamiento de los usuarios. Gracias a un enfoque ágil, basado en pequeños ciclos, pruebas controladas (A/B testing) y entrega continua, se pudo introducir cambios significativos sin incurrir en altos costos técnicos ni retrasos operativos.

Desde esta perspectiva, el costo del cambio en agilidad no se elimina, pero se gestiona inteligentemente: se reconoce, se

reduce su impacto y se incorpora como parte del proceso de construcción del software.

El desarrollo ágil no solo admite el cambio, sino que lo convierte en una ventaja competitiva. Al integrar prácticas técnicas sólidas, interacción frecuente con los usuarios y ciclos de trabajo cortos, las metodologías ágiles logran reducir el impacto económico y técnico del cambio. En lugar de temerle, los equipos ágiles se preparan para adaptarse a él de manera eficiente y continua.

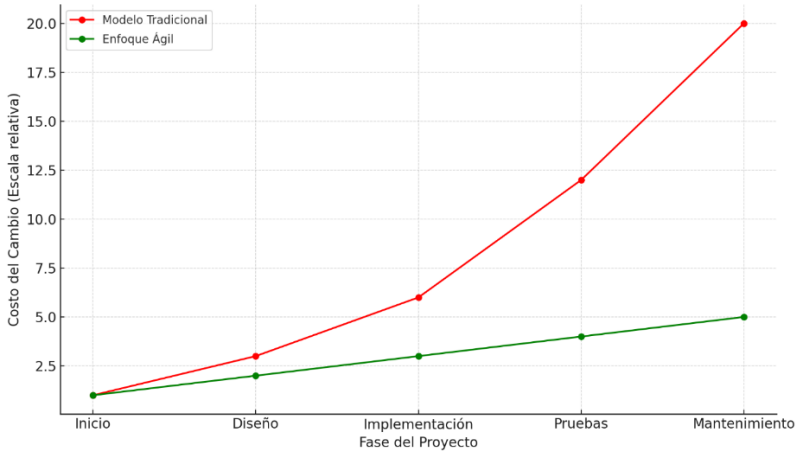

Figura 3
Comparación el costo del cambio
Nota. Elaboración propia

Resumen del capítulo 2

Este capítulo presentó los fundamentos conceptuales, metodológicos y prácticos del desarrollo ágil de software. Se abordó primero el contexto histórico que dio origen al enfoque ágil, destacando las limitaciones de los modelos tradicionales de desarrollo y la necesidad de una respuesta más flexible ante entornos de alta incertidumbre y cambio constante. En este marco, se analizó el surgimiento del Manifiesto Ágil en 2001 y sus aportes en cuanto a valores y principios que priorizan la colaboración, la entrega temprana de valor y la adaptabilidad.

Luego se introdujo el concepto de procesos ágiles, entendidos como estructuras flexibles e iterativas para construir software mediante ciclos cortos, feedback continuo y mejora progresiva. Se enfatizó que estos procesos no son recetas cerradas, sino marcos que permiten adaptarse al contexto organizacional. A través del caso de ING Bank, se ilustró cómo la adopción de procesos ágiles puede transformar incluso organizaciones tradicionales y altamente reguladas.

Posteriormente se describieron con profundidad tres enfoques ágiles ampliamente utilizados: Scrum, Extreme Programming (XP) y Kanban. De Scrum se explicaron sus roles, eventos, artefactos y ciclos de trabajo, ejemplificados con el modelo organizacional de Spotify. En XP, se abordaron prácticas técnicas como TDD, pair programming e integración continua, ilustradas con el caso del desarrollo de JUnit. En el caso de Kanban, se destacaron su enfoque en la visualización del flujo de trabajo, la limitación de tareas en curso y su aplicación exitosa en la BBC.

También se presentó una comparación entre los tres enfoques, resaltando sus similitudes, diferencias y contextos de aplicación. Se evidenció que no existe un enfoque único válido, sino que cada uno puede ser más adecuado según el tipo de proyecto, la madurez del equipo y los objetivos del producto.

Finalmente, se exploró el concepto de costo del cambio en entornos ágiles. A diferencia de los modelos tradicionales, donde el cambio se vuelve más costoso con el tiempo. En el enfoque ágil este impacto se reduce considerablemente gracias a prácticas como la entrega continua, la retroalimentación temprana y el diseño evolutivo. Se explicó cómo los equipos ágiles integran el cambio como parte natural del proceso, permitiendo construir productos más adaptables y centrados en el usuario.

Preguntas de revisión

1. ¿Cuáles fueron las principales limitaciones de los modelos tradicionales que motivaron el surgimiento del enfoque ágil? Explica con un caso real.
2. ¿Qué valores y principios propone el Manifiesto Ágil y cómo influyen en la organización del trabajo en los equipos de desarrollo?
3. Explique qué es un proceso ágil. ¿De qué forma se diferencia de un proceso tradicional de desarrollo de software?
4. Compare Scrum, XP y Kanban en cuanto a estructura, enfoque y adaptabilidad. ¿En qué contextos es más recomendable aplicar cada uno?
5. ¿Cuál es el propósito de limitar el trabajo en curso (WIP) en Kanban y qué impacto tiene en la productividad del equipo?
6. ¿Qué prácticas técnicas hacen que XP sea considerado una metodología orientada a la calidad del código? Menciona al menos tres y justifica su utilidad.
7. ¿Por qué el enfoque ágil permite mantener bajo el costo del cambio incluso en etapas avanzadas del proyecto? Sustenta tu respuesta con un ejemplo.

Autoevaluación

Parte A: Verdadero o Falso

Indique si las siguientes afirmaciones son verdaderas (V) o falsas (F).

1. El Manifiesto Ágil fue creado en 2005 como respuesta a la crisis financiera de proyectos tecnológicos. ( )
2. En XP, el cliente debe estar involucrado permanentemente en el proceso de desarrollo. ( )
3. Scrum requiere que los equipos trabajen en iteraciones fijas llamadas sprints. ( )
4. Kanban es más adecuado para procesos con entregas continuas y mantenimiento. ( )

5. El Product Owner es responsable de facilitar las reuniones diarias del equipo Scrum. ( )

Parte B: Selección múltiple

Seleccione la opción correcta para cada pregunta.

1. ¿Cuál de los siguientes elementos no es un artefacto de Scrum?
    a) Product Backlog
    b) Sprint Retrospective
    c) Sprint Backlog
    d) Incremento
2. ¿Cuál es una práctica central de Extreme Programming (XP)?
    e) Priorización de tareas mediante story points
    f) Pruebas guiadas por código
    g) Programación en parejas
    h) Gestión visual del flujo de trabajo
3. ¿Qué técnica se utiliza en Kanban para limitar la multitarea y aumentar el enfoque del equipo?
    i) Velocity tracking
    j) Work in Progress (WIP) limit
    k) Sprint Planning
    l) Daily stand-ups

Parte C: Relacione las columnas

Relacione cada enfoque ágil de la columna A con su característica distintiva en la columna B.

Columna A	**Columna B**
a) Scrum	( ) Optimiza el flujo mediante tableros visuales y límites WIP
b) Kanban	( ) Se enfoca en la calidad del código con prácticas técnicas

c) Extreme Programming	( ) Divide el trabajo en sprints con entregas frecuentes

Ejercicio de aplicación final

"Tres equipos, un producto: ¿qué enfoque ágil elegir?"

Contexto del caso:

La empresa FinTech Solutions ha sido contratada para desarrollar una plataforma digital de servicios financieros orientada a jóvenes emprendedores que necesitan acceder rápidamente a créditos, gestionar pagos en línea y recibir asesoría financiera. El proyecto es estratégico, ya que el cliente desea lanzar el producto en seis meses para aprovechar una ventana de mercado.

La dirección de proyectos decidió dividir el trabajo en tres equipos multidisciplinarios, cada uno con responsabilidades diferenciadas:

- Equipo A: se centra en el diseño de interfaces y la experiencia de usuario (UI/UX). Deben validar constantemente prototipos con usuarios reales y adaptarse a cambios frecuentes en los requisitos estéticos y de interacción.
- Equipo B: desarrolla la lógica central del sistema, que incluye autenticación de usuarios, manejo de transacciones financieras y reglas de negocio complejas. Sus entregas deben ser altamente confiables y cumplir regulaciones de seguridad.
- Equipo C: da soporte a sistemas legados que deben integrarse con la nueva plataforma (ej. bases de datos históricas y un sistema de facturación). Su trabajo está marcado por interrupciones constantes debido a incidentes urgentes, lo que dificulta mantener una planificación rígida.

La gerencia busca adoptar enfoques ágiles para:

- Aumentar la entrega de valor en ciclos cortos.

- Mejorar la comunicación entre equipos con ritmos y necesidades diferentes.
- Reducir el tiempo de salida al mercado sin sacrificar calidad ni cumplimiento normativo.

Sin embargo, no existe claridad sobre qué enfoque ágil (Scrum, XP, Kanban u otro híbrido) sería más adecuado para cada equipo, considerando sus particularidades.

Actividades para desarrollar:

a) Analice las características de cada equipo. ¿Qué enfoque ágil (Scrum, XP o Kanban) sería más adecuado para cada uno? Justifique su elección.
b) Explique cómo el enfoque propuesto para el Equipo B puede ayudar a manejar cambios frecuentes en reglas de negocio y requerimientos técnicos.
c) ¿De qué manera puede el Equipo C beneficiarse de una gestión visual del flujo de trabajo?
d) ¿Qué elementos del Manifiesto Ágil deberían ser compartidos por los tres equipos, sin importar el enfoque particular que utilicen?

Objetivos del ejercicio:

- Aplicar de manera contextualizada los conceptos clave del capítulo.
- Desarrollar habilidades de análisis comparativo entre metodologías ágiles.
- Promover la reflexión crítica sobre adaptabilidad de procesos ágiles según las características del equipo y del proyecto.

Capítulo 3: Principios, requisitos y tipos de modelado

Introducción del capítulo

En el proceso de desarrollo de software, una de las etapas más críticas es la correcta identificación y especificación de los requerimientos del sistema. La ingeniería de requisitos no solo permite entender qué necesita el cliente, sino también establecer un puente sólido entre los problemas del mundo real y las soluciones tecnológicas que se construirán. Este capítulo se centra en el estudio de los procesos fundamentales para capturar y refinar esos requerimientos: la indagación, el análisis, la validación y, finalmente, el modelado, que permite representarlos gráficamente para una mejor comprensión, comunicación y verificación.

A través del uso de técnicas modernas, se abordan distintos enfoques de modelado, incluyendo los diagramas UML más utilizados en la industria, como los casos de uso, diagramas de clases, de flujo y de comportamiento. Además, se introduce el modelado orientado a aplicaciones web, cuyas particularidades: como la navegación no lineal, la interactividad y la integración con servicios en línea, exigen herramientas visuales específicas como mapas de navegación, wireframes y prototipos de interfaz.

El propósito de este capítulo es brindar al lector los fundamentos conceptuales y prácticos para especificar y modelar de manera efectiva los requerimientos del software, fortaleciendo la etapa de análisis del proceso de desarrollo y preparando el camino para un diseño robusto y coherente.

Objetivos de aprendizaje

Al finalizar este capítulo, el lector será capaz de:

- Comprender los fundamentos de la ingeniería de requerimientos en el proceso de desarrollo de software.
- Identificar y aplicar técnicas de indagación, análisis y validación de requerimientos.

- Elaborar modelos de requerimientos utilizando distintos tipos de diagramas UML.
- Diferenciar los enfoques de modelado estructural, de clases, de flujo y de comportamiento.
- Aplicar herramientas y técnicas de modelado adaptadas a las particularidades de las aplicaciones web.

Esquema del capítulo

3.1 Ingeniería de Requerimientos
- 3.1.1 Indagación de requerimientos
- 3.1.2 Elaboración de modelos de requerimientos
- 3.1.3 Análisis de los requerimientos
- 3.1.4 Validación de los requerimientos
- 3.1.5 Modelado de requerimientos mediante casos de uso

3.2 Modelado Basado en Escenarios y Modelos UML
- 3.2.1 Modelado de Datos
- 3.2.2 Modelado Basado en Clases
- 3.2.3 Modelado Orientado al Flujo
- 3.2.4 Modelo de Comportamiento

3.3 Modelado de Requerimientos para Webapps
- 3.3.1 Características específicas de aplicaciones web
- 3.3.2 Modelos de navegación y usabilidad
- 3.3.3 Adaptación de UML para aplicaciones web

Preguntas guía

- ¿Por qué es esencial una correcta ingeniería de requerimientos en el desarrollo de software?
- ¿Qué técnicas permiten indagar, analizar y validar los requerimientos de un sistema?
- ¿Qué tipos de diagramas UML existen y cómo se utilizan en el modelado de software?
- ¿Qué diferencias existen entre el modelado estructural, de flujo, de clases y de comportamiento?
- ¿Cómo se modelan los requerimientos en aplicaciones web y qué herramientas se utilizan?

3.1 Ingeniería de Requerimientos

3.1.1 Indagación de Requerimientos

Fuentes de información para la identificación de requisitos

La identificación adecuada de los requerimientos de un sistema es esencial para asegurar que el software desarrollado cumpla con las expectativas del cliente y se ajuste al contexto operativo. Para ello, se debe recurrir a diversas fuentes de información, que permiten recopilar datos relevantes desde múltiples perspectivas.

Entre las fuentes más comunes se encuentran:

- Usuarios finales: Proveen información directa sobre las necesidades, expectativas y problemas que enfrentan en su interacción con sistemas similares o procesos manuales actuales.
- Clientes o patrocinadores: Representan la voz del negocio. Su enfoque suele estar orientado a los objetivos estratégicos, restricciones de tiempo y presupuesto, y metas de rentabilidad.
- Expertos del dominio: Son personas con profundo conocimiento del área específica del sistema, como médicos en el caso de software clínico, o abogados en sistemas legales. Su participación es clave para garantizar que los requerimientos estén alineados con prácticas profesionales y normativas vigentes.
- Documentación existente: Manuales de procesos, sistemas anteriores, bases de datos, registros y políticas organizacionales proporcionan un contexto formal que ayuda a definir funcionalidades requeridas o identificar restricciones técnicas y legales.
- Reglamentos y normas: Algunas aplicaciones deben cumplir requisitos legales, estándares de calidad o normativas técnicas (como ISO, IEEE o leyes nacionales), que también constituyen fuentes de requerimientos no funcionales importantes.

- Observación directa y simulaciones de procesos: El análisis de la operativa real mediante observación in situ, grabaciones o simulaciones puede revelar flujos de trabajo que no siempre son verbalizados por los usuarios.
- Sistemas existentes o competidores: Estudiar plataformas ya implementadas o soluciones del mercado puede ofrecer ideas para funcionalidades, mejoras o diferenciación estratégica.
- Utilizar una combinación equilibrada de estas fuentes es esencial para evitar la dependencia de una única visión del sistema y garantizar una perspectiva integral. Además, esta diversidad ayuda a descubrir requerimientos latentes que el usuario no siempre es capaz de expresar verbalmente, pero que se hacen evidentes en la práctica

Técnicas de recolección de requerimientos

Una vez identificadas las fuentes de información, es necesario aplicar técnicas sistemáticas que permitan obtener los requerimientos de manera estructurada y comprensible. La elección de la técnica adecuada depende del tipo de proyecto, la disponibilidad de los actores clave, y el nivel de formalidad deseado. A continuación, se describen las técnicas más utilizadas en la ingeniería de requerimientos:

- Entrevistas. Son encuentros estructurados o semiestructurados con stakeholders (usuarios, clientes, expertos del dominio) para obtener información detallada. Las entrevistas pueden ser abiertas (más exploratorias) o cerradas (con preguntas definidas), y son especialmente útiles para entender necesidades específicas y expectativas individuales.
- Cuestionarios o encuestas. Se aplican cuando se requiere obtener información de un grupo amplio de personas, especialmente cuando los actores están distribuidos geográficamente. Permiten recopilar

datos de manera rápida y uniforme, aunque limitan la posibilidad de profundización.

- Talleres de requerimientos (workshops). Son sesiones colaborativas que reúnen a múltiples partes interesadas con el objetivo de identificar y consensuar requerimientos. Promueven la interacción, la negociación y la resolución temprana de discrepancias, y son muy efectivos en entornos ágiles.
- Observación directa. Consiste en estudiar cómo los usuarios interactúan actualmente con el sistema o con los procesos manuales que se desean automatizar. Esta técnica permite descubrir requerimientos tácitos y detectar ineficiencias o comportamientos recurrentes.
- Prototipado. Involucra la creación rápida de representaciones preliminares del sistema (como pantallas o diagramas de flujo) para que los usuarios visualicen cómo funcionará. El prototipado permite validar y refinar requerimientos con base en retroalimentación directa.
- Análisis de documentos. Se examinan manuales, informes, reglamentos, contratos y sistemas heredados para extraer requerimientos explícitos e implícitos. Es particularmente útil cuando se busca reemplazar o integrar sistemas existentes.
- Técnicas etnográficas y de inmersión. Se emplean en entornos complejos, donde los ingenieros de software se integran temporalmente a los contextos de uso para comprender la dinámica organizacional y las necesidades reales, incluso aquellas que los usuarios no saben cómo expresar.

El uso combinado de estas técnicas permite abordar los requerimientos desde distintos ángulos, aumentando la confiabilidad del resultado final. Además, facilita la participación activa de los usuarios en el proceso, lo que mejora la aceptación del sistema y reduce el riesgo de fallas por especificaciones incompletas o erróneas.

Documentación y gestión de requisitos

Una vez que los requerimientos han sido recolectados mediante las técnicas apropiadas, es fundamental documentarlos de manera clara, coherente y verificable. La documentación de requerimientos actúa como un contrato entre los stakeholders y el equipo de desarrollo, y constituye la base para el diseño, la implementación y las pruebas del sistema.

Documentación de requisitos

La Especificación de Requisitos del Software (SRS, por sus siglas en inglés) es el documento estándar en el que se consignan los requerimientos funcionales y no funcionales. Este documento debe ser:

- Completo: incluir todos los comportamientos esperados y restricciones.
- Consistente: sin contradicciones internas.
- Comprensible: usando lenguaje claro, evitando ambigüedades.
- Modificable: estructurado para facilitar su actualización.
- Rastreable: cada requerimiento debe poder asociarse a su fuente y a su implementación posterior.

Los requerimientos funcionales describen qué debe hacer el sistema (por ejemplo, "el sistema permitirá a los usuarios registrarse mediante correo electrónico"), mientras que los no funcionales detallan cómo debe comportarse (por ejemplo, "el sistema debe responder en menos de 3 segundos").

Gestión de requisitos

La gestión de requerimientos es el proceso continuo de rastrear, priorizar, validar y controlar los cambios sobre los requerimientos a lo largo del ciclo de vida del proyecto. Involucra:

- Priorización: no todos los requerimientos tienen el mismo valor; algunos son críticos, otros deseables. Se

utilizan técnicas como MoSCoW (Must, Should, Could, Won't).
- Rastreo bidireccional: permite vincular cada requerimiento con su origen (usuarios, documentos, leyes) y su destino (casos de prueba, módulos de código, manuales).
- Control de cambios: todo cambio en los requerimientos debe ser analizado en términos de impacto técnico, económico y temporal. Se suelen registrar en sistemas de control de versiones o herramientas especializadas.
- Versionamiento: mantener un historial claro de versiones de los requerimientos ayuda a gestionar su evolución sin perder trazabilidad.

Para apoyar estas tareas, existen herramientas como Jama Connect, IBM DOORS, RequisitePro, Jira con complementos como Confluence, y otras plataformas orientadas a la ingeniería de software y la documentación colaborativa.

Una gestión efectiva de requerimientos es crítica para prevenir desviaciones de alcance, asegurar la satisfacción del cliente y reducir los costos de retrabajo, especialmente en proyectos de gran envergadura o de larga duración.

3.1.2 Elaboración de modelos de requerimientos

Especificación de requisitos funcionales y no funcionales

La especificación de requerimientos es una actividad clave que transforma la información recolectada en representaciones claras y estructuradas, comprensibles tanto para los stakeholders como para el equipo técnico. Dentro de esta especificación se distinguen dos grandes tipos de requerimientos: funcionales y no funcionales.

Requisitos funcionales

Los requerimientos funcionales describen los servicios, funciones y comportamientos que el sistema debe ofrecer. Responden a la pregunta: *¿Qué debe hacer el sistema?* Algunos ejemplos comunes incluyen:

"El sistema permitirá al usuario registrar una nueva cuenta mediante correo electrónico o redes sociales."

"El sistema debe enviar una notificación automática al administrador cuando se supere el límite de almacenamiento."

Estos requerimientos suelen especificarse utilizando verbos de acción, y se pueden organizar por módulos, procesos de negocio, o casos de uso.

Requisitos no funcionales

Los requerimientos no funcionales establecen condiciones o restricciones sobre cómo debe comportarse el sistema. Son esenciales para asegurar la calidad, la seguridad, el rendimiento y la mantenibilidad del software. Algunas categorías frecuentes son:

- Rendimiento: tiempos de respuesta, concurrencia de usuarios.
- Seguridad: cifrado, autenticación, permisos de acceso.
- Fiabilidad: disponibilidad del sistema, tolerancia a fallos.
- Usabilidad: facilidad de aprendizaje, accesibilidad.
- Compatibilidad: soporte de navegadores, dispositivos o sistemas operativos.
- Por ejemplo:
- "El sistema debe cargar todas las vistas en menos de 2 segundos en condiciones normales de red."
- "El sistema debe cumplir con el estándar WCAG 2.1 para accesibilidad."

Una buena práctica es establecer criterios de aceptación medibles para los requerimientos no funcionales, lo cual facilita su verificación y prueba posterior.

Herramientas y técnicas de modelado de requerimientos

Una vez que los requerimientos han sido especificados, el siguiente paso es representarlos de forma visual o estructurada mediante modelos, lo que permite analizarlos,

validarlos y comunicarlos de manera más eficaz entre los miembros del equipo de desarrollo y los stakeholders.

El modelado de requerimientos cumple varios objetivos:

- Identificar ambigüedades u omisiones.
- Validar la lógica del sistema antes del diseño técnico.
- Favorecer la comunicación entre usuarios y desarrolladores.
- Establecer una base formal para el diseño y las pruebas.

Técnicas de modelado más comunes

1. Casos de uso. Permiten representar las interacciones entre actores externos (usuarios, sistemas) y el sistema, organizadas por funcionalidades clave. Utilizan diagramas que muestran las relaciones y flujos generales. Son especialmente útiles en metodologías orientadas a objetos o ágiles.
2. Modelado de procesos. Incluye diagramas de flujo, diagramas de actividades (UML), y BPMN (Business Process Model and Notation), que representan las tareas, decisiones y secuencias que conforman un proceso de negocio.
3. Diagramas de clases (UML). Representan las entidades del sistema (clases), sus atributos, métodos y relaciones. Son fundamentales para definir la estructura lógica del sistema.
4. Diagramas de flujo de datos (DFD). Muestran cómo la información fluye entre procesos, almacenes de datos y entidades externas. Son adecuados para el modelado funcional, especialmente en proyectos que no usan orientación a objetos.
5. Historias de usuario. Utilizadas en metodologías ágiles, representan requerimientos en forma de frases breves desde la perspectiva del usuario. Por ejemplo: "Como cliente, quiero ver el estado de mi pedido para saber si ya fue despachado."
6. Wireframes y prototipos de interfaces. Muestran la apariencia y funcionalidad de las interfaces gráficas, lo que

ayuda a validar requerimientos relacionados con usabilidad, navegación e interacción.

Herramientas digitales más utilizadas

- Enterprise Architect: para modelado UML avanzado.
- Lucidchart / Draw.io: para diagramas colaborativos en línea.
- Balsamiq / Figma: para prototipado de interfaces.
- Visual Paradigm: combina modelado UML con análisis de requerimientos.
- Jira / Confluence: para documentación viva de requerimientos integrados a gestión ágil.

El uso de herramientas adecuadas mejora la trazabilidad, fomenta la colaboración entre equipos distribuidos y garantiza una documentación perdurable, adaptable y alineada con estándares industriales.

3.1.3 Análisis de los requerimientos

Clasificación y priorización de requisitos

Una vez recolectados y modelados, los requerimientos deben ser analizados sistemáticamente para evaluar su viabilidad, identificar redundancias o conflictos, y establecer su relevancia para el sistema. El análisis de requerimientos permite organizar la información de forma que facilite la toma de decisiones en etapas posteriores del desarrollo.

Clasificación de requisitos

La clasificación tiene como objetivo agrupar los requerimientos según su naturaleza, propósito o impacto. Las categorías más comunes incluyen:

- Funcionales vs. no funcionales: Según lo que el sistema debe hacer o cómo debe comportarse.
- Requerimientos esenciales vs. deseables: Los esenciales son imprescindibles para el funcionamiento mínimo del sistema; los deseables agregan valor, pero pueden ser omitidos si hay limitaciones.

- Requerimientos del usuario vs. del sistema: Los primeros expresan necesidades generales del usuario; los segundos son una traducción técnica más detallada.
- Requerimientos legales, regulatorios o contractuales: Aquellos que surgen de leyes, normativas o acuerdos firmados con el cliente.

Esta clasificación ayuda a determinar el nivel de criticidad de cada requerimiento, su relación con otras partes del sistema y los criterios de aceptación.

Priorización de requisitos

El proceso de priorización permite a los equipos de desarrollo tomar decisiones estratégicas sobre qué requerimientos deben implementarse primero, especialmente cuando los recursos son limitados o el tiempo es ajustado. A continuación, se detallan tres enfoques ampliamente utilizados:

1. Método MoSCoW

El método MoSCoW es una técnica de priorización simple pero poderosa, utilizada sobre todo en entornos ágiles como DSDM (Dynamic Systems Development Method). Su nombre es un acrónimo formado a partir de las iniciales de sus cuatro categorías:

- Must Have (Debe tener). Requerimientos críticos e imprescindibles. El sistema no podrá funcionar sin ellos y su ausencia implica un fracaso del proyecto. Suelen estar relacionados con funciones básicas, criterios legales o necesidades contractuales.

  Ejemplo: "El sistema debe permitir iniciar sesión mediante credenciales válidas".

- Should Have (Debería tener). Requerimientos importantes, pero no esenciales. Su exclusión no impide la funcionalidad principal del sistema, pero sí puede reducir su calidad, eficiencia o satisfacción del usuario.

Ejemplo: "El sistema debería enviar un correo de confirmación después del registro".

- Could Have (Podría tener). Requerimientos deseables, con bajo impacto si no se implementan. Son elementos que mejoran la experiencia del usuario o agregan valor adicional, pero que pueden posponerse si hay limitaciones de tiempo o presupuesto.

  Ejemplo: "El sistema podría incluir una guía interactiva para usuarios nuevos".

- Won't Have (No tendrá por ahora). Requerimientos que no se abordarán en esta versión, aunque pueden ser considerados en futuras actualizaciones. Esta categoría ayuda a controlar el alcance y evitar la expansión descontrolada (scope creep).

  Ejemplo: "Integración con redes sociales no será considerada en la primera entrega".
  Este método fomenta el diálogo entre usuarios y desarrolladores, y permite revaluar prioridades en función del avance del proyecto.

2. Análisis de valor y costo (Cost-Value Approach)

Este enfoque consiste en comparar cada requerimiento en función de dos dimensiones clave:

- Valor (beneficio): mide el impacto positivo que el requerimiento tiene para el negocio, los usuarios o el sistema.
- Costo (esfuerzo): evalúa el esfuerzo técnico, tiempo y recursos requeridos para implementarlo.

Con esta información, los requerimientos se pueden representar en una matriz valor-costo. Lo ideal es priorizar aquellos que ofrecen alto valor a bajo costo. Las decisiones también pueden depender del tipo de producto (mínimo viable, versión beta, versión estable).

Tabla 6

Clasificación de funcionalidades según valor y costo

Categoría	**Acción recomendada**
Alto valor, bajo costo	Alta prioridad. Se implementan primero.
Alto valor, alto costo	Evaluar retorno de inversión
Bajo valor, bajo costo	Posponer o usar como mejoras opcionales
Bajo valor, alto costo	Descartar, salvo requerimientos obligatorios

Nota. Elaboración propia

Esta técnica es muy útil para justificar decisiones ante stakeholders en términos cuantificables.

3. Matrices de priorización

Las matrices permiten representar los requerimientos de forma visual y estructurada, clasificándolos con base en dos o más criterios (como urgencia e impacto, o valor y riesgo). Una de las formas más comunes es la matriz de impacto-urgencia, en la que se traza un eje X (por ejemplo, urgencia) y un eje Y (impacto), y se posicionan los requerimientos en el plano:

Tabla 7

Matriz de priorización basada en urgencia e impacto

	Urgencia alta	**Urgencia baja**
Impacto alto	Alta prioridad inmediata	Planificar a mediano plazo
Impacto bajo	Revisar, puede ser crítica	Opcional o descartable

Nota. Elaboración Propia

Este tipo de matrices ayuda a visualizar rápidamente qué requerimientos tienen más peso estratégico, facilitando la

toma de decisiones en reuniones con equipos interdisciplinarios.

Resolución de conflictos en requisitos de software

Durante el proceso de recolección y análisis de requerimientos, es común que surjan conflictos entre los diferentes actores del proyecto. Estos pueden deberse a visiones opuestas sobre las prioridades del sistema, ambigüedades en la interpretación de necesidades, restricciones técnicas, o incluso a la resistencia al cambio por parte de algunos usuarios. La resolución adecuada de estos conflictos es esencial para garantizar la coherencia, la aceptación del sistema y el cumplimiento de los objetivos del proyecto.

Tipos comunes de conflicto

- Conflictos de prioridad: Un grupo de usuarios considera que cierto requerimiento es crítico, mientras otro lo ve como innecesario.
- Conflictos de definición: Diferentes partes interesadas tienen interpretaciones distintas sobre un mismo requerimiento.
- Conflictos técnicos: Los requerimientos deseados no son compatibles entre sí, o exceden las capacidades del sistema o del equipo.
- Conflictos organizacionales: Surgen cuando los requerimientos de un área afectan negativamente los intereses de otra.

"Los conflictos de requerimientos son inevitables en proyectos de software complejos; lo importante es detectarlos tempranamente y tratarlos como una oportunidad para el consenso y la mejora del producto final."
Ian Sommerville (2011), Ingeniería del Software, 9ª ed., p. 150

Técnicas para resolver conflictos

1. Negociación facilitada. Involucra a un facilitador neutral que guía la conversación entre las partes para identificar intereses comunes y llegar a acuerdos. Se

enfoca en ganar-ganar, evitando la imposición de una sola visión.
2. Modelado alternativo. Crear dos versiones distintas de un mismo requerimiento y analizarlas con criterios objetivos (costo, riesgo, usabilidad, alineación estratégica). Esto permite tomar decisiones basadas en datos y evidencia, no solo en percepciones.
3. Priorización colaborativa. Aplicar técnicas como MoSCoW, votación ponderada o análisis de valor con la participación activa de todos los stakeholders, permitiendo una decisión democrática y justificada.
4. Revisión iterativa de prototipos. Mediante prototipos funcionales o wireframes, se prueba cada opción en escenarios reales o simulados, lo que permite identificar qué versión resuelve mejor el problema.
5. Entrevistas o consultas con expertos del dominio. Cuando el conflicto es técnico o especializado, se recurre a profesionales con experiencia comprobada en el área para orientar la toma de decisiones.

En palabras de Pressman (2014): "La resolución de conflictos de requerimientos exige equilibrio entre diplomacia, conocimiento técnico y comprensión del dominio del problema" (*Ingeniería del Software. Un enfoque práctico*, 7ª ed., p. 169).

Buenas prácticas

- Documentar todos los conflictos detectados, sus causas y la decisión adoptada.
- Justificar cada resolución en términos de valor de negocio, impacto técnico o satisfacción del usuario.
- Revisar las decisiones durante el desarrollo para validar su vigencia.
- Mantener la comunicación abierta entre los actores clave del proyecto.

La capacidad de resolver conflictos de manera constructiva no solo previene retrasos y sobrecostos, sino que también fortalece la relación entre los equipos técnicos y los usuarios, y mejora la calidad final del software.

3.1.4 Validación de los requerimientos

La validación de requerimientos es la etapa en la que se revisa si los requerimientos recolectados y documentados representan con fidelidad las necesidades de los usuarios y si son consistentes, completos y viables. A diferencia de la verificación, que responde a la pregunta *¿Se ha construido el sistema correctamente?*, la validación se enfoca en *¿Se está construyendo el sistema correcto?*

La validación temprana es crucial para evitar errores costosos en etapas posteriores. Según estudios citados por Boehm (2000), el costo de corregir un error en requerimientos puede ser hasta 100 veces mayor si se detecta en producción que si se corrige en la fase de análisis.

"La validación de requerimientos no es un acto único, sino un proceso continuo de diálogo con los usuarios, de revisión crítica y de mejora progresiva del entendimiento del sistema deseado."
— Pressman y Maxim (2015), Ingeniería del Software: Un enfoque práctico, p. 154

Métodos de validación y verificación

Existen múltiples enfoques para validar requerimientos, que pueden combinarse según el contexto del proyecto:

1. Revisiones estructuradas. Reuniones formales donde un grupo de expertos revisa los documentos de requerimientos en busca de ambigüedades, inconsistencias, omisiones o errores. Suele usarse una lista de verificación.
2. Prototipos y simulaciones. Permiten mostrar a los usuarios representaciones funcionales o visuales del sistema esperado. Son especialmente útiles para validar requerimientos de interfaz, flujo y usabilidad.
3. Casos de prueba de validación. Diseñar pruebas basadas en requerimientos permite verificar si cada uno puede ser implementado y evaluado. Si un requerimiento no es comprobable, debe reformularse.

4. Walkthroughs
   Recorrido informal y guiado del documento de requerimientos, usualmente con el cliente o con representantes clave del usuario, que permite identificar rápidamente discrepancias conceptuales.
5. Listas de control (checklists). Utilizadas para garantizar que todos los atributos de calidad estén presentes: claridad, rastreabilidad, coherencia, completitud, factibilidad, entre otros.

Sommerville (2011) señala que "la validación efectiva depende más de la participación activa del usuario que de la sofisticación técnica del modelo" (*Ingeniería del Software*, 9.ª ed., p. 160).

Evaluación de consistencia y completitud

- Consistencia:
  Se refiere a que no existan contradicciones entre requerimientos. Por ejemplo, un requerimiento que indica que un informe debe generarse automáticamente cada día, y otro que requiere confirmación manual antes de generarlo, estarían en conflicto.
- Completitud:
  Se asegura de que todas las funcionalidades necesarias estén cubiertas. Un sistema de pagos que especifica cómo registrar los pagos pero omite la anulación o reembolso carecería de completitud.

Las herramientas modernas como RequisitePro, Jama Connect o IBM DOORS ayudan a rastrear inconsistencias, validar dependencias y mantener un control riguroso sobre los cambios y el estado de validación de cada requerimiento.

3.1.5 Modelado de requerimientos mediante casos de uso

Los casos de uso son una técnica ampliamente utilizada en la ingeniería de requerimientos para describir interacciones entre los usuarios (actores) y el sistema, enfocándose en las

funcionalidades que el sistema debe ofrecer desde una perspectiva externa.

Un caso de uso describe un conjunto de escenarios que producen un resultado observable y valioso para un actor. Su propósito es capturar requerimientos funcionales de forma estructurada y comprensible, incluso para usuarios no técnicos.

"Los casos de uso permiten estructurar los requerimientos funcionales como historias de interacción, lo que facilita su validación con los usuarios y su trazabilidad hacia el diseño y las pruebas."-*Cockburn, A. (2000). Writing Effective Use Cases.*

Elementos de un caso de uso:

- Actor: Persona, sistema o entidad externa que interactúa con el sistema.
- Caso de uso: Función que el sistema ofrece al actor.
- Relaciones:
    - Asociación: Actor participa en un caso de uso.
    - Inclusión (include): Reutilización de un caso de uso común.
    - Extensión (extend): Comportamiento opcional que se activa bajo ciertas condiciones.

Ejemplo: Sistema de pedidos en línea

Actores: Cliente

Casos de uso:

- Realizar pedido
- Consultar productos
- Registrar cuenta
- Ver estado del pedido

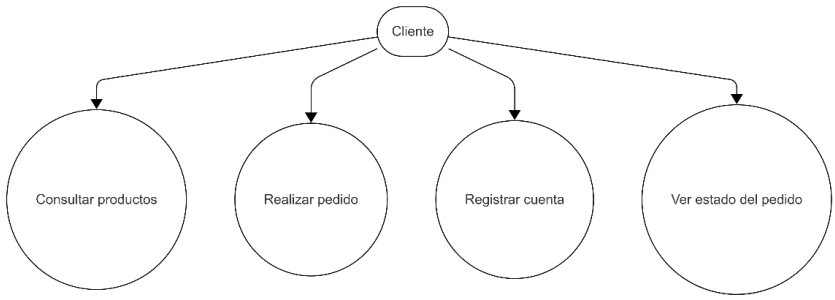

Figura 4
Caso de uso
Nota. Elaboración propia

3.2 Modelado basado en escenarios y modelos UML

3.2.1 Modelado de datos

Es una técnica esencial para representar la estructura lógica de la información que manejará un sistema. Su objetivo es identificar y organizar los datos clave, sus atributos, relaciones y restricciones, lo que sirve de base tanto para la implementación de bases de datos como para el diseño conceptual del sistema.

Diagramas entidad-relación (ER)

El modelo entidad-relación (ER) fue propuesto por Peter Chen en 1976 y se ha convertido en uno de los estándares para representar datos estructurados. Este modelo utiliza tres componentes fundamentales:

- Entidades: Representan objetos del mundo real o conceptos importantes del dominio (por ejemplo, Cliente, Producto, Pedido).
- Atributos: Son las propiedades de las entidades (por ejemplo, nombre del cliente, fecha del pedido).
- Relaciones: Indican cómo se asocian las entidades entre sí (por ejemplo, un Cliente realiza muchos Pedidos).

Los diagramas ER permiten visualizar de forma intuitiva las dependencias entre los elementos de datos. Las relaciones

pueden tener diferentes cardinalidades, como uno a uno (1:1), uno a muchos (1:N) o muchos a muchos (M:N), lo cual influye directamente en el diseño posterior de la base de datos.

"El modelado de datos define el universo de discurso del sistema: aquello que se quiere conocer, registrar y transformar" *Date, C. J. (2004). An Introduction to Database Systems, 8th ed.*

3.2.2 Modelado basado en clases

El modelado basado en clases es una técnica fundamental en el análisis y diseño orientado a objetos. Permite representar las entidades del sistema como clases, identificando sus atributos, métodos (comportamientos) y relaciones con otras clases. Este tipo de modelado no solo ayuda a comprender la estructura del sistema, sino que también sirve como base para la implementación en lenguajes como Java, C#, Python, entre otros.

"El diagrama de clases es el plano estructural del software orientado a objetos; define lo que el sistema es antes de que se defina lo que hará."*-Booch, Rumbaugh y Jacobson (2005), The Unified Modeling Language User Guide*

Definición y construcción de clases

Una clase es una abstracción que describe un conjunto de objetos con características y comportamientos comunes. Cada clase tiene:

- Nombre
- Atributos: propiedades o datos que caracterizan a la clase.
- Métodos: operaciones o acciones que puede realizar.

Por ejemplo:

Clase: Cliente

Atributos: nombre, correo, teléfono

Métodos: registrar(), autenticar()

Relaciones entre clases en UML

Las clases pueden relacionarse de varias maneras. UML define los siguientes tipos de relaciones:

Asociación

Es una relación estructural básica que indica que una clase está conectada con otra. La asociación puede tener una cardinalidad para indicar cuántas instancias de una clase pueden estar relacionadas con instancias de otra.

Ejemplo:

Un Cliente realiza uno o más Pedidos

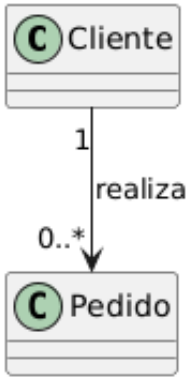

Figura 5
Asociación
Nota. Elaboración propia

Agregación (relación “tiene un” débil)

La agregación es una forma especializada de asociación que representa una relación "todo/parte". La clase “todo” contiene a las clases “parte”, pero las partes pueden existir de forma independiente.

Ejemplo:

Una Universidad tiene muchas facultades, pero estas podrían existir por separado

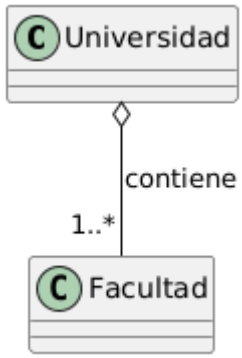

Figura 6
Agregación
Nota. Elaboración propia

Composición (relación "tiene un" fuerte)

La composición es una forma más fuerte de agregación. Las partes no pueden existir sin el todo. Si se elimina la clase contenedora, también se eliminan sus componentes.

Ejemplo:

Una Casa tiene Habitaciones. Si se destruye la casa, las habitaciones dejan de existir

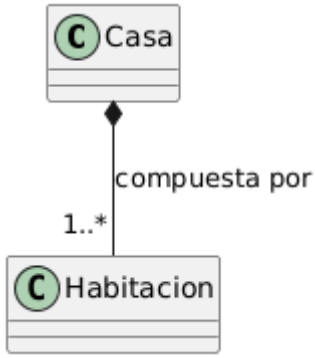

Figura 7
Composición
Nota. Elaboración propia

Herencia o generalización

Una clase hija hereda atributos y comportamientos (métodos) de una clase padre. Representa una relación “es un tipo de”.

Ejemplo:

Empleado es una clase general, Profesor y Administrador son clases específicas que heredan de ella.

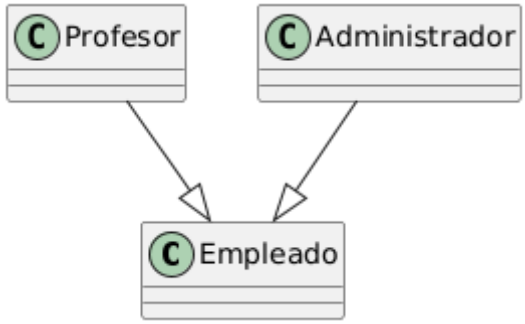

Figura 8
Herencia
Nota. Elaboración propia

Dependencia

Indica que una clase usa a otra de manera temporal. Es una relación débil, no estructural, generalmente a través de parámetros o variables locales.

Ejemplo:

Una clase Factura usa una clase EmailService para enviar notificaciones, pero no tiene una relación permanente con ella.

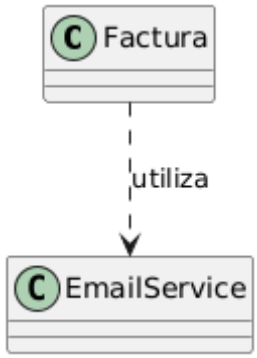

Figura 9
Dependencia
Nota. Elaboración propia

3.2.3 Modelado orientado al flujo

El modelado orientado al flujo representa cómo fluye la información a través de un sistema, mostrando las transformaciones que experimentan los datos y cómo interactúan con los actores externos. Este enfoque resulta particularmente útil en las etapas de análisis para comprender procesos existentes y diseñar nuevos sistemas.

"Los modelos orientados al flujo no muestran estructuras, sino transformaciones: son mapas de movimiento, no de organización."- *Yourdon & DeMarco (1979), Structured Analysis and System Specification.*

Los DFD se estructuran en niveles jerárquicos para representar un sistema con distintos grados de detalle. Los niveles 0 y 1 son los más comunes al inicio del análisis.

En un ejemplo conceptual:

Supongamos un sistema de pedidos en línea:

En un sistema de pedidos en línea, el proceso comienza cuando el cliente realiza un pedido a través de la interfaz del sistema; dicha solicitud es registrada automáticamente en la base de datos para su almacenamiento y seguimiento, y posteriormente, el sistema genera y emite una factura

correspondiente al pedido efectuado, la cual puede ser enviada al cliente como comprobante de la transacción

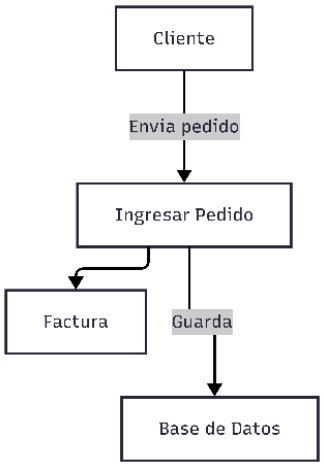

Figura 10
Modelado orientado al flujo Nivel 0
Nota. Elaboración propia

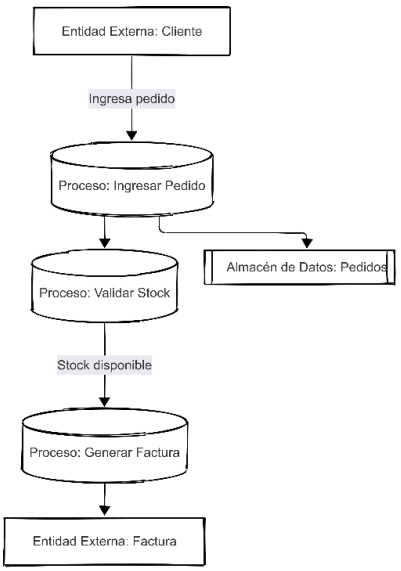

Figura 11
Modelado orientado al flujo Nivel 1
Nota. Elaboración propia

3.2.4 Modelo de comportamiento

El modelo de comportamiento en ingeniería de software describe cómo interactúan los objetos del sistema a lo largo del tiempo y cómo responden a eventos o estímulos. A diferencia de los modelos estructurales (como clases o datos), el modelo de comportamiento se enfoca en el flujo de control, los mensajes y la evolución del estado del sistema.

"Los modelos de comportamiento permiten simular, prever y validar las dinámicas internas del software antes de su implementación."-*Sommerville, I. (2011). Ingeniería del Software, 9.ª ed., p. 190*

Diagramas de secuencia

Representan la interacción temporal entre objetos o actores mediante el intercambio de mensajes. Son ideales para describir casos de uso dinámicos, así por ejemplo:

Realizar pedido: Un cliente accede al sistema de pedidos en línea y realiza una solicitud para comprar uno o varios productos. El sistema valida la información ingresada y, si todo está correcto, procede a almacenar el pedido en la base de datos. A continuación, el sistema genera una factura electrónica correspondiente al pedido, que es registrada y puede ser enviada al cliente. Finalmente, el sistema envía una confirmación al cliente notificando que el pedido ha sido recibido exitosamente.

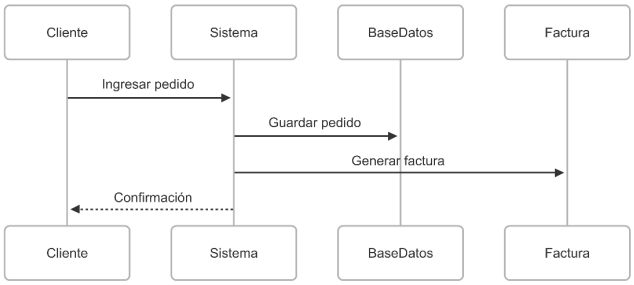

Figura 12
Diagramas de secuencia: Realizar pedido
Nota. Elaboración propia

Diagramas de estados

Representan los estados posibles de un objeto o sistema y los eventos que causan transiciones entre estados. Se usan para modelar sistemas reactivos, como interfaces, flujos de autorización o procesos transaccionales.

Ejemplo: Ciclo de vida de un pedido

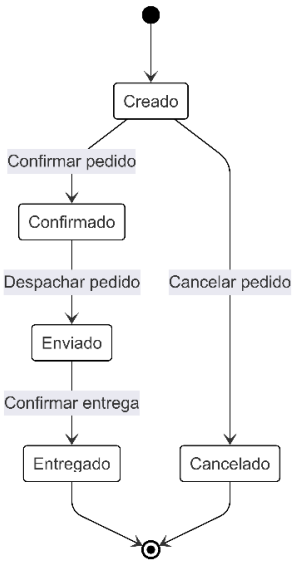

Figura 23
Diagramas de estados: Ciclo de vida de un pedido
Nota. Elaboración propia

3.3 Modelado de requerimientos para webapps

Las aplicaciones web (WebApps) presentan características únicas que requieren enfoques específicos para el modelado de requerimientos. A diferencia del software tradicional, las WebApps se ejecutan en entornos distribuidos, interactúan constantemente con el usuario y están sujetas a cambios frecuentes. Esto implica considerar aspectos como la navegación, usabilidad, compatibilidad multiplataforma y la interacción asincrónica.

"El modelado de WebApps requiere una perspectiva centrada en la experiencia del usuario y en los flujos de navegación, sin perder de vista los aspectos funcionales y estructurales del sistema."-*Pressman & Lowe (2009), Web Engineering: A Practitioner's Approach*

3.1.1 Características específicas de aplicaciones web

Entre los aspectos que diferencian a las WebApps se encuentran:

- Alta interactividad: el usuario espera respuestas rápidas y experiencias fluidas.
- Interfaces dinámicas: se requiere modelar el comportamiento visual (carga parcial, validación instantánea).
- Navegación no lineal: los usuarios pueden acceder a funciones desde múltiples caminos.
- Accesibilidad desde distintos dispositivos y navegadores.
- Seguridad web: autenticación, validación del lado del cliente y servidor, protección contra inyecciones o accesos no autorizados.
- Integración con servicios externos (APIs, pasarelas de pago, redes sociales).

Estas características deben reflejarse desde los requerimientos, tanto funcionales como no funcionales, y guiar el diseño de modelos adaptados al entorno web.

3.3.2 Modelos de navegación y usabilidad

Para representar el comportamiento y flujo de las WebApps, se utilizan modelos centrados en la experiencia del usuario, como:

- Mapas de navegación (site maps): muestran la estructura jerárquica o conectividad entre páginas o módulos.
- Diagramas de interacción: similares a los diagramas de secuencia, pero enfocados en eventos UI, transiciones y respuestas visuales.
- Wireframes: esquemas de interfaz que muestran la distribución de los elementos visuales y los flujos de interacción.
- User flows: describen paso a paso el recorrido del usuario desde una necesidad hasta su resolución.

"Una buena experiencia de usuario comienza con un modelado claro de cómo se moverá el usuario a través de la aplicación, qué verá y cómo interactuará."-*Garrett, J. (2011). The Elements of User Experience.*

Ejemplo: Flujo de usuario para una compra en WebApp

1. Usuario accede a la página principal.
2. Navega hasta el catálogo de productos.
3. Añade un producto al carrito.
4. Ingresa sus datos y confirma la compra.
5. Recibe una notificación de confirmación.

Este recorrido puede representarse mediante un user flow o un diagrama de estados aplicando transiciones en función de acciones del usuario.

3.3.3 Adaptación de UML para aplicaciones web

Aunque UML no fue diseñado exclusivamente para aplicaciones web, puede adaptarse para modelar:

- Casos de uso web específicos: por ejemplo, "Registrarse mediante redes sociales", "Subir imagen con validación en vivo".
- Diagramas de actividades con decisiones condicionales en formularios.
- Diagramas de componentes y despliegue para representar la arquitectura web.
- Diagramas de clases que incluyan entidades web como Usuario, Sesión, Vista, Controlador, siguiendo el patrón MVC.

También se puede complementar con herramientas como:

- Figma o Adobe XD para wireframes interactivos.
- Lucidchart o Draw.io para mapas de navegación.
- BPMN para modelar flujos complejos de usuario.

Resumen del capítulo 3

En este capítulo se abordaron los fundamentos de la ingeniería de requerimientos como una etapa crítica en el desarrollo de software. Se estudiaron las técnicas de

indagación, documentación y gestión de requisitos, diferenciando entre requerimientos funcionales y no funcionales, y presentando herramientas modernas para su modelado. Asimismo, se exploraron los enfoques de análisis, priorización y resolución de conflictos. Posteriormente, se introdujeron los distintos tipos de modelos visuales: desde diagramas estructurales (como entidad-relación y clases), hasta modelos de comportamiento (como secuencias y estados). También se detallaron los métodos específicos para modelar requerimientos en aplicaciones web, incorporando herramientas como wireframes, mapas de navegación y flujos de usuario. Todo este conocimiento permite representar de forma clara y precisa lo que un sistema debe hacer y cómo debe comportarse, sentando las bases para una construcción robusta, coherente y centrada en el usuario

Preguntas de revisión

1. ¿Cuáles son las principales fuentes de información para la recolección de requerimientos?
2. ¿Qué diferencia existe entre un requerimiento funcional y uno no funcional?
3. ¿Qué técnicas se utilizan para validar y verificar requerimientos?
4. ¿Cómo se representa un sistema mediante un diagrama de flujo de datos (DFD)?
5. ¿Qué tipos de relaciones existen entre clases en UML?
6. ¿Cuáles son las ventajas de los diagramas de estado para modelar comportamiento?
7. ¿Qué particularidades deben considerarse al modelar requerimientos para WebApps?

Autoevaluación

Parte A: Verdadero o Falso

Indique si las siguientes afirmaciones son verdaderas (V) o falsas (F).

1. La ingeniería de requerimientos se limita únicamente a recolectar necesidades de los usuarios, sin considerar restricciones técnicas ni de negocio. ( )
2. Los requerimientos funcionales describen lo que el sistema debe hacer, mientras que los no funcionales especifican condiciones de calidad y restricciones. ( )
3. La indagación de requerimientos es una actividad estática que no necesita la participación activa de los stakeholders. ( )
4. Los modelos basados en escenarios, como los casos de uso, permiten representar interacciones entre usuarios y el sistema. ( )
5. El modelado de datos se enfoca en representar la estructura de la información, mientras que el modelado de clases organiza las entidades, atributos y relaciones en un contexto orientado a objetos. ( )
6. El modelado de flujo describe cómo la información o las actividades se desplazan dentro del sistema, facilitando la identificación de procesos clave. ( )
7. Los modelos de comportamiento, como los diagramas de estados, ayudan a entender la dinámica del sistema frente a distintos eventos. ( )
8. La validación de requerimientos busca asegurar que lo documentado corresponde a las verdaderas necesidades del cliente y puede ser implementado. ( )
9. En el modelado de requerimientos para aplicaciones web (webapps) es suficiente utilizar diagramas de datos; no se requiere considerar navegación ni experiencia de usuario. ( )
10. Una especificación de requerimientos clara y validada contribuye directamente a reducir costos y errores en las etapas posteriores del ciclo de vida del software. ( )

Parte B: Selección múltiple

Seleccione la opción correcta para cada pregunta.

1. ¿Cuál de los siguientes elementos corresponde a un requerimiento no funcional?
   a) El usuario puede registrarse con su correo electrónico
   b) El sistema debe permitir agregar productos al carrito
   c) El sistema debe responder en menos de 2 segundos
   d) El cliente puede consultar su historial de compras
2. ¿Qué técnica es más adecuada para obtener requerimientos cuando se desea explorar a fondo las necesidades de un usuario clave?
   a) Cuestionario
   b) Observación
   c) Entrevista semiestructurada
   d) Prototipado
3. En un diagrama de clases UML, ¿qué tipo de relación indica una herencia entre clases?
   a) Asociación
   b) Composición
   c) Dependencia
   d) Generalización
4. ¿Qué herramienta de modelado es más adecuada para representar el flujo de interacción cronológica entre objetos?
   a) Diagrama de clases
   b) Diagrama de casos de uso
   c) Diagrama de secuencia
   d) Diagrama de flujo de datos
5. En un modelo de comportamiento, ¿qué representa una transición en un diagrama de estados?
   a) Un mensaje entre actores
   b) Un cambio de condición en la base de datos
   c) Un paso en un algoritmo
   d) Un evento que hace pasar de un estado a otro

6. ¿Cuál de los siguientes enfoques es más específico para representar el flujo de navegación en una aplicación web?
   a) Diagrama de clases
   b) Wireframe
   c) Mapa de navegación
   d) Diagrama de estados

Parte C: Relacione las columnas

Relaciona cada término del grupo A con su definición correspondiente del grupo B.

Grupo A	**Grupo B**
a) DFD	( ) Representación estructural de objetos y sus relaciones
b) Caso de uso	( ) Muestra interacciones con el sistema desde la perspectiva del usuario
c) Diagrama de clases	( ) Describe el flujo de información entre procesos y almacenes
d) Diagrama de estados	( ) Representa los cambios en la condición interna de un objeto

Ejercicio de aplicación final

"Plataforma web de venta de entradas para eventos culturales"

La Fundación CulturaViva organiza conciertos, obras de teatro y exposiciones en varias ciudades. Busca desarrollar una aplicación web que permita a los usuarios descubrir eventos, consultar fechas y localidades disponibles, comprar entradas con pago en línea y recibir el ticket digital por correo. La salida al mercado debe ser rápida, con una versión mínima utilizable en 8 semanas, y cumpliendo requisitos legales de protección de datos.

Actores y roles

- Usuario invitado: navega eventos y consulta disponibilidad sin comprar.
- Usuario registrado: compra, descarga y gestiona sus entradas.
- Administrador de eventos: crea/edita eventos, horarios, aforos y precios.
- Pasarela de pago: procesa transacciones y devuelve estado de pago.
- Sistema de correo: envía confirmaciones y tickets digitales (PDF/QR).

Alcance (versión inicial)

- Catálogo navegable por ciudad, fecha y categoría.
- Selección de evento, fecha, tipo de entrada y cantidad.
- Compra con tarjeta a través de pasarela de pago externa.
- Emisión de ticket digital con código QR y envío por correo.
- Backoffice mínimo para alta/edición de eventos y control de aforo.

Supuestos y restricciones

- Solo una moneda y un país en esta versión.
- Un único proveedor de pasarela de pago.
- Políticas de reembolso: solo antes del inicio del evento.
- Cumplimiento de protección de datos personales.
- El QR se valida en puerta con un lector externo ya disponible.

Datos de apoyo (para modelar)

- Evento: id, título, descripción corta, ciudad, lugar, categoría, fechas/funciones, aforo por función, precios por tipo (general/estudiante/VIP).
- Usuario: id, nombre, email, hash de contraseña.
- Orden: id, usuario, función, ítems (tipo de entrada, cantidad, precio), total, estado pago (pendiente/aprobado/rechazado), timestamp.

- Ticket: id, orden, QR, estado (vigente/usado/anulado).

Actividades para desarrollar

1. Requerimientos funcionales y no funcionales
Elabore una lista de al menos 10 requerimientos, diferenciando explícitamente:

- Funcionales (qué hace el sistema).
- No funcionales (calidad, restricciones, seguridad, rendimiento, usabilidad, portabilidad, mantenibilidad). Sugerencia: incluya disponibilidad, tiempos de respuesta, accesibilidad web, manejo de errores en pago, logging y privacidad.

2. Diagrama de casos de uso
Modele las funcionalidades clave para los actores definidos. Al menos:

- Buscar/browsing de eventos, gestionar cuenta, comprar entradas, gestionar eventos, procesar pago, emitir ticket, enviar confirmación. Indique inclusiones/extensiones cuando corresponda (p. ej., autenticar usuario, calcular total, aplicar descuento).

3. Diagrama de clases
Diseñe las entidades principales con atributos y relaciones. Considere clases como:

- Evento, Función, Precio, Usuario, Carrito/Orden, ItemOrden, Ticket, Pago, Pasarela, Administrador. Incluya multiplicidades y, cuando aplique, composiciones.

4. Diagrama de flujo de datos (DFD nivel 1) del proceso de compra
Represente procesos, almacenes y flujos para:

- Seleccionar función y entradas, calcular total, enviar orden a pasarela, recibir estado, generar tickets, actualizar aforo y enviar correo. Identifique claramente

las entradas/salidas con los externos Pasarela y Sistema de correo.

Objetivos del ejercicio

- Aplicar los conocimientos del capítulo sobre ingeniería de requerimientos y modelado en un caso práctico de desarrollo de software.
- Diferenciar entre requerimientos funcionales y no funcionales y documentarlos de forma clara.
- Representar las funcionalidades clave del sistema mediante un diagrama de casos de uso.
- Diseñar un diagrama de clases que muestre las entidades principales y sus relaciones.
- Elaborar un diagrama de flujo de datos (nivel 1) que refleje el proceso de compra de entradas.
- Desarrollar habilidades de análisis, abstracción y modelado de sistemas a partir de un escenario realista.

Capítulo 4: Concepciones y Tipos de Diseño

Introducción del capítulo

El diseño de software constituye una de las fases más importantes dentro del ciclo de vida del desarrollo, pues representa el punto de transición entre la definición de lo que se desea construir (requerimientos) y cómo se construirá. En este capítulo se estudiarán los principios fundamentales del diseño, incluyendo la modularidad, el refinamiento, la independencia funcional y el ocultamiento de información, así como los conceptos orientados a objetos que sustentan muchas de las metodologías modernas. A continuación, se analizarán los modelos arquitectónicos más utilizados, como la arquitectura en capas, microservicios y serverless, y se revisarán los criterios para seleccionar una arquitectura adecuada. El capítulo también explora los patrones de diseño, que ofrecen soluciones reutilizables para problemas comunes de software, y analiza los distintos tipos de diseño según el nivel de abstracción o especialización (componentes, dominios, experiencia de usuario). Finalmente, se abordará el diseño de interfaces, entendidas como el puente entre la lógica del sistema y la interacción humana, incorporando principios de usabilidad y evaluación centrada en el usuario.

Objetivos de aprendizaje

Al finalizar este capítulo, el lector será capaz de:

- Comprender los principios y fundamentos que rigen el diseño de software.
- Aplicar conceptos como modularidad, refinamiento y ocultamiento para construir software robusto y mantenible.
- Analizar arquitecturas adecuadas para distintos tipos de sistemas.
- Identificar patrones de diseño en contextos reales.
- Diferenciar tipos de diseño según el nivel y enfoque del sistema.

- Integrar principios de usabilidad en el diseño de interfaces de usuario.

Esquema del capítulo

4.1 Conceptos de diseño de software
  - 4.1.1 Principios fundamentales del diseño
  - 4.1.2 Independencia funcional y modularidad
  - 4.1.3 Refinamiento y ocultamiento de información
  - 4.1.4 Conceptos orientados a objetos aplicados al diseño

4.2 Arquitectura del software
  - 4.2.1 Modelos arquitectónicos y estilos de arquitectura
  - 4.2.2 Arquitectura en capas, microservicios y serverless
  - 4.2.3 Factores clave en la selección de una arquitectura

4.3 Patrones de diseño
  - 4.3.1 Introducción a los patrones de diseño
  - 4.3.2 Patrones de creación, estructurales y de comportamiento

4.4 Tipos de diseño en ingeniería de software
  - 4.4.1 Diseño a nivel de componentes
  - 4.4.2 Diseño basado en dominios específicos
  - 4.4.3 Diseño centrado en la experiencia del usuario

4.5 Diseño de interfaces de usuario
  - 4.5.1 Principios de diseño de interfaces
  - 4.5.2 Evaluación y pruebas de usabilidad
  - 4.5.3 Integración entre el diseño de interfaz y la lógica de negocio

Preguntas guía

- ¿Qué principios garantizan un diseño de software comprensible y sostenible?
- ¿Cuáles son las ventajas de aplicar modularidad e independencia funcional?
- ¿Qué tipos de arquitecturas existen y cómo influyen en la escalabilidad del sistema?

- ¿Qué problemas resuelven los patrones de diseño y cómo se aplican?
- ¿Cómo influye el diseño de la interfaz en la experiencia del usuario final?

4.1 Conceptos de diseño de software

4.1.1 Principios fundamentales del diseño de software

El diseño de software busca convertir los requerimientos definidos durante el análisis en una solución técnica estructurada, comprensible y mantenible. Para lograrlo, los ingenieros de software deben aplicar una serie de principios fundamentales, que actúan como guías conceptuales para tomar decisiones de diseño que equilibren funcionalidad, calidad, escalabilidad y facilidad de mantenimiento.

"El diseño de software es un proceso de toma de decisiones progresivas, cuyo objetivo es estructurar el sistema de forma que maximice la calidad y minimice el costo de evolución."- *Pressman & Maxim (2015). Ingeniería de Software. Un enfoque práctico, p. 327*

Principios operativos del diseño de software

1. Simplicidad
   Un diseño simple es más fácil de entender, probar y modificar. La simplicidad reduce el riesgo de errores y mejora la mantenibilidad. Esto no significa que el sistema sea limitado, sino que debe evitar complejidades innecesarias.
2. Modularidad
   El sistema debe estar dividido en componentes independientes (módulos), donde cada uno cumple una función específica. Esta separación favorece el desarrollo colaborativo, la reutilización y la localización de errores.
3. Cohesión y acoplamiento

4. Alta cohesión: cada módulo debe encargarse de una única responsabilidad. Bajo acoplamiento: los módulos deben tener mínimas dependencias entre sí. Esta combinación mejora la estabilidad del diseño frente a cambios.
5. Abstracción
   El diseño debe centrarse en representar lo esencial, ocultando los detalles internos del funcionamiento de los módulos. Permite pensar a diferentes niveles de complejidad sin perder claridad.
6. Encapsulamiento y ocultamiento de información. Cada módulo debe exponer solo lo necesario para interactuar con otros. Sus detalles internos deben permanecer inaccesibles para proteger la integridad del sistema y facilitar futuras modificaciones.
7. Separación de responsabilidades. Cada parte del sistema debe encargarse de un tipo específico de tarea: interfaz de usuario, lógica de negocio, acceso a datos, etc. Esto da origen a arquitecturas como MVC (Modelo–Vista–Controlador).
8. Principio de anticipación al cambio. Un buen diseño debe prever la posibilidad de evolución. Se deben construir componentes que puedan ampliarse o modificarse sin afectar otras partes del sistema.
9. Reutilización. El diseño debe favorecer el uso de componentes ya existentes (internos o de terceros) mediante librerías, APIs o frameworks, para reducir tiempos y costos.

Estos principios, aplicados conjuntamente, ayudan a diseñar sistemas que no solo cumplen con los requerimientos funcionales, sino que también son sostenibles a lo largo del tiempo. A medida que el sistema evoluciona, estos principios se convierten en la base para soportar cambios de manera controlada y económica.

"Diseñar bien es anticiparse al cambio con estructuras que puedan adaptarse sin romperse."- *Robert C. Martin (2003),*

Agile Software Development Principles, Patterns, and Practices.

4.1.2 Independencia funcional y modularidad

Uno de los principios clave del diseño de software es la independencia funcional, que se refiere a la capacidad de cada módulo o componente del sistema para ejecutar una tarea específica de forma autónoma, sin depender en exceso de otros módulos. Esta característica se materializa a través de la modularidad, que consiste en descomponer el sistema en partes más pequeñas, manejables y comprensibles.

Según Pressman y Maxim (2015), "un módulo independiente debe poseer alta cohesión interna y bajo acoplamiento externo", lo que significa que sus elementos internos están fuertemente relacionados entre sí, mientras que su dependencia con el resto del sistema es mínima.

¿Por qué es importante la modularidad?

- Facilita la comprensión: cada módulo se puede analizar en aislamiento.
- Favorece la reutilización: los módulos pueden ser adaptados o incorporados en otros proyectos.
- Mejora la mantenibilidad: al reducir dependencias, los cambios en un módulo tienen menor impacto en el resto del sistema.
- Permite el desarrollo paralelo: diferentes equipos pueden trabajar en distintos módulos de manera simultánea.

Como señala Sommerville (2011), "la modularidad permite que el sistema se construya a través de unidades separadas que luego se integran, lo que reduce el riesgo global del proyecto y mejora la calidad del software" (p. 225).

Criterios para evaluar la independencia de un módulo

Existen dos métricas principales:

- Cohesión: mide el grado en que los elementos de un módulo contribuyen a un propósito único y bien definido. Se busca alta cohesión.
    - Por ejemplo, un módulo que solo gestiona operaciones de autenticación debe tener funciones relacionadas exclusivamente con iniciar sesión, validar credenciales, cerrar sesión, etc.
- Acoplamiento: evalúa el grado de dependencia entre módulos. Se busca bajo acoplamiento para evitar que un cambio en un módulo obligue a modificar otros.
    - Un módulo de autenticación no debería depender directamente de un módulo de facturación.

"Un diseño modular no solo mejora la estructura del sistema, sino que lo vuelve más resiliente ante cambios, facilitando su evolución con bajo impacto en la arquitectura general."- *Gamma et al. (1994), Design Patterns: Elements of Reusable Object-Oriented Software*.

En resumen, diseñar con independencia funcional y modularidad no es solo una cuestión de estructura, sino una estrategia para gestionar la complejidad y garantizar la sostenibilidad del software a largo plazo. Es un principio esencial en cualquier arquitectura moderna, desde microservicios hasta aplicaciones móviles.

4.1.3 Refinamiento y ocultamiento de información

Dos de los principios más importantes para lograr un diseño robusto, evolutivo y mantenible en ingeniería de software son el refinamiento y el ocultamiento de información. Ambos permiten construir sistemas complejos por capas de abstracción y proteger las partes internas de los componentes frente a cambios externos.

Refinamiento

El refinamiento es un proceso gradual mediante el cual un diseño se desarrolla desde una forma general y abstracta hasta una representación más detallada y concreta. En otras palabras, es un desplazamiento descendente del nivel de abstracción hacia el nivel de implementación.

"Refinar significa dividir un proceso complejo en pasos más simples, detallando progresivamente lo que se necesita para llevar a cabo una función."-Pressman & Maxim (2015), p. 331

El refinamiento permite que el diseñador comience con una visión de alto nivel, por ejemplo: "Validar acceso del usuario", y luego la descomponga en acciones más específicas como: "Verificar existencia del usuario", "Comprobar contraseña", "Registrar intento fallido".

Ventajas del refinamiento:

- Permite gestionar la complejidad mediante pasos progresivos.
- Facilita la planificación del desarrollo por fases o capas.
- Mejora la comunicación entre diseñadores, analistas y programadores.

Como señala Sommerville (2011), "el refinamiento es esencial para mantener la trazabilidad entre el diseño y los requerimientos iniciales, especialmente en sistemas grandes y multidisciplinarios" (p. 228).

Ocultamiento de información

El ocultamiento de información es el principio por el cual un módulo expone únicamente lo que es necesario para su interacción externa, mientras oculta sus detalles internos de implementación. Este concepto fue introducido por David Parnas en 1972 y es un pilar de la programación estructurada y orientada a objetos.

"El ocultamiento de información permite diseñar módulos que sean independientes en su implementación, pero consistentes en su interfaz."-Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules.

Este principio se traduce en:

- Encapsulamiento de datos y lógica interna.
- Interfaz bien definida para acceder a las funciones del módulo.
- Reducción del impacto de cambios: si cambia el algoritmo interno, no es necesario modificar los módulos que lo utilizan.

Por ejemplo, un módulo que realiza cálculos financieros puede ocultar su fórmula interna y exponer solo un método como calcular Impuesto(ingreso). Si el algoritmo de cálculo cambia, otros módulos que usan esa función no necesitan ser modificados.

Relación entre ambos

- El refinamiento trabaja sobre el contenido del diseño, detallándolo paso a paso.
- El ocultamiento de información trabaja sobre la exposición del diseño, limitando lo visible al exterior.

Ambos principios complementan el modularidad: uno la estructura en niveles de detalle, el otro la protege de interdependencias innecesarias.

“Diseñar es un acto dual: detallar lo que importa y esconder lo que puede cambiar.”-Gamma et al. (1994), Design Patterns.

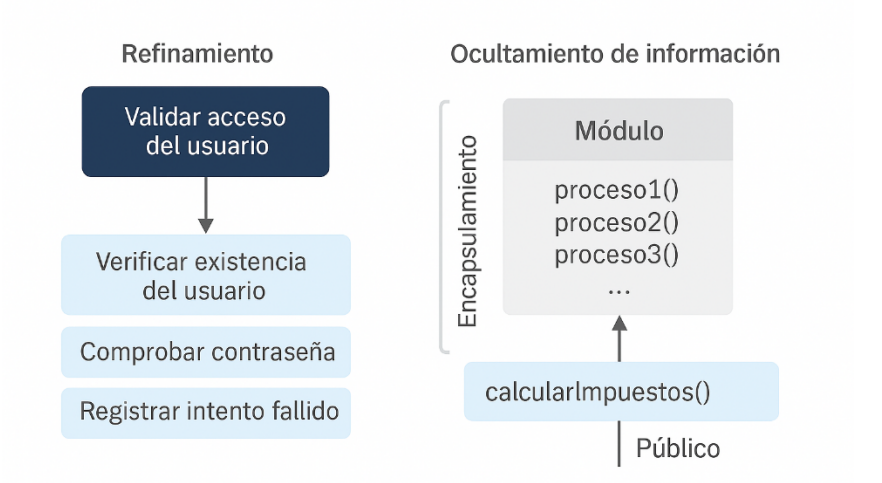

Figura 34
Principios de refinamiento y ocultamiento de información
Nota. Elaboración propia

4.1.4 Conceptos orientados a objetos aplicados al diseño

El paradigma orientado a objetos (POO) ha transformado la manera en que se diseña software, permitiendo una representación más cercana al mundo real mediante entidades denominadas objetos, que encapsulan estado (atributos) y comportamiento (métodos). Aplicar los conceptos orientados a objetos en la fase de diseño mejora la organización interna del sistema, facilita su reutilización, promueve la modularidad y fortalece la mantenibilidad del código.

"El diseño orientado a objetos ofrece una forma de construir software alineado con la forma en que los humanos perciben el mundo: como una colección de objetos que interactúan entre sí."-*Booch, Rumbaugh & Jacobson (2005). The Unified Modeling Language User Guide.*

Principales conceptos aplicados al diseño

1. Clase y objeto

- Una clase es una plantilla que define un conjunto de atributos y métodos comunes.
- Un objeto es una instancia concreta de una clase.
- En el diseño, se definen clases como Cliente, Factura, Producto, y se especifica su interacción mediante diagramas UML.

2. Encapsulamiento

- Encierra los datos y los métodos dentro del objeto.
- Evita el acceso directo desde el exterior, promoviendo el ocultamiento de información.
- Solo se expone una interfaz pública para interactuar con el objeto.

Por ejemplo, una clase Cuenta Bancaria puede tener un método retirar(monto) pero no permitir el acceso directo al saldo.

3. Herencia

- Permite crear clases derivadas que heredan atributos y comportamientos de una clase base.
- Favorece la reutilización y reduce la duplicación de código.
- Se usa para establecer jerarquías: Empleado puede ser superclase de Gerente y Asistente.

4. Polimorfismo

- Capacidad de un objeto de responder de manera distinta a un mismo mensaje según su tipo.
- Se logra mediante la sobrecarga (mismo nombre, distintos parámetros) o sobreescritura (mismo método redefinido en subclases).
- Por ejemplo, el método calcular Sueldo() puede comportarse distinto en Empleado y Gerente.

5. Abstracción

- Permite centrarse en las características esenciales de un objeto, ignorando los detalles irrelevantes.
- Se implementa mediante interfaces o clases abstractas.
- Ideal para representar roles o comportamientos genéricos (ej. Autenticable, Imprimible).

"Los conceptos fundamentales del diseño orientado a objetos permiten construir software robusto, extensible y expresivo, alineado con la lógica del dominio del problema."-*Gamma et al. (1994), Design Patterns*.

4.2 Arquitectura del software

La arquitectura del software representa la estructura general de un sistema en términos de sus componentes principales, sus relaciones y las reglas que rigen su organización e interacción. Es la base sobre la cual se construye y evoluciona el sistema, y tiene un impacto directo en su rendimiento, escalabilidad, seguridad y mantenibilidad.

"La arquitectura define el marco estructural dentro del cual se toman las decisiones de diseño detallado y se coordinan las responsabilidades de desarrollo."-Bass, Clements & Kazman (2012). Software Architecture in Practice.

4.2.1 Modelos arquitectónicos y estilos de arquitectura

Un modelo arquitectónico describe un patrón organizativo general para un sistema. Existen varios estilos arquitectónicos reconocidos que proporcionan soluciones estructuradas y reutilizables para distintos contextos.

Estilos comunes:

- Arquitectura en capas. El sistema se divide en capas horizontales con responsabilidades específicas (presentación, lógica de negocio, acceso a datos). Facilita la separación de responsabilidades y la escalabilidad vertical.

- Cliente-servidor. Los clientes solicitan servicios al servidor, que centraliza la lógica y el acceso a datos. Común en aplicaciones web tradicionales.
- Microservicios. El sistema se compone de múltiples servicios independientes, cada uno con su propia lógica y base de datos. Facilita el despliegue independiente, el escalado selectivo y la tolerancia a fallos.
- Arquitectura orientada a eventos (EDA). Basada en la producción y consumo de eventos. Los componentes reaccionan ante eventos sin necesidad de conocer su origen. Útil en sistemas distribuidos y asincrónicos.
- Pipe & Filter (Tubería y filtros). Se usa en procesamiento secuencial de datos. Cada filtro transforma la información y la pasa al siguiente. Común en compiladores o procesamiento de señales.

"Los estilos arquitectónicos no son recetas, sino marcos conceptuales que permiten razonar sobre el diseño a gran escala."- *Shaw & Garlan (1996). Software Architecture: Perspectives on an Emerging Discipline*.

4.2.2 Arquitectura en capas, microservicios y serverless

Arquitectura en capas

La arquitectura en capas es uno de los estilos más tradicionales y ampliamente utilizados en el diseño de sistemas de software. Este modelo organiza el sistema en estratos horizontales, donde cada capa cumple una función específica y se comunica únicamente con las capas adyacentes. La división típica incluye al menos tres capas: presentación, lógica de negocio y acceso a datos. Por ejemplo, en una aplicación web de comercio electrónico, la capa de presentación se encarga de mostrar la interfaz gráfica del sitio (HTML, CSS, JavaScript), la lógica de negocio gestiona operaciones como el cálculo de totales o la validación de pagos, y la capa de datos maneja las conexiones a la base de datos para almacenar pedidos, clientes y productos.

Este modelo promueve la separación de responsabilidades, permitiendo que los desarrolladores trabajen de forma independiente en cada capa, lo que facilita el mantenimiento, la prueba y la evolución del sistema. Por ejemplo, si se requiere cambiar el motor de base de datos (como pasar de MySQL a PostgreSQL), es probable que solo se afecte la capa de datos. No obstante, uno de los errores comunes al implementar este estilo es permitir acoplamientos cruzados, es decir, que una capa de presentación acceda directamente a la base de datos, rompiendo así la arquitectura lógica.

Sommerville (2011) destaca que "las arquitecturas en capas son particularmente útiles en sistemas de información donde las responsabilidades están naturalmente distribuidas en distintos niveles de abstracción" (p. 250). Son ideales para sistemas monolíticos bien estructurados, aplicaciones empresariales internas y plataformas web con flujos estables y bien definidos.

Arquitectura de microservicios

La arquitectura de microservicios representa un enfoque moderno y descentralizado para el diseño de sistemas, en el cual la funcionalidad se distribuye entre múltiples servicios pequeños, autónomos y especializados. Cada microservicio está orientado a resolver una única responsabilidad de negocio, posee su propia lógica interna, puede tener su propia base de datos y se comunica con otros servicios a través de interfaces bien definidas, como APIs REST, gRPC o colas de mensajes.

Por ejemplo, en una plataforma de compras como Amazon, es común que haya microservicios independientes para gestionar el catálogo de productos, el carrito de compras, el sistema de pagos, y la gestión de usuarios. Esto permite que distintos equipos trabajen de forma autónoma sobre cada servicio, escalen sus componentes de forma selectiva y desplieguen cambios sin afectar a todo el sistema.

Sin embargo, este estilo también introduce complejidades adicionales. La coordinación entre servicios requiere

mecanismos robustos de comunicación, monitoreo, gestión de errores y seguridad. Además, las transacciones distribuidas —como una operación de pago que involucra múltiples servicios— deben manejarse con patrones como sagas o compensaciones.

Según Newman (2015), "los microservicios ofrecen una vía escalable para construir sistemas grandes, pero su adopción requiere un cambio de mentalidad arquitectónica y organizacional" (*Building Microservices*, O'Reilly). Es una solución ideal para sistemas que requieren alta disponibilidad, despliegue continuo, escalabilidad horizontal y evolución independiente por componentes.

Arquitectura Serverless (FaaS - Function as a Service)

La arquitectura serverless o FaaS (Function as a Service) es un paradigma emergente que permite a los desarrolladores implementar funciones de software que se ejecutan en la nube sin tener que gestionar servidores ni preocuparse por el escalado de infraestructura. En este modelo, cada unidad funcional del sistema se representa como una función que responde a eventos (por ejemplo, una solicitud HTTP, un archivo subido, un mensaje de una cola) y que se ejecuta bajo demanda en un entorno gestionado por el proveedor de nube (como AWS Lambda, Google Cloud Functions o Azure Functions).

Un ejemplo común es una función que genera miniaturas de imágenes al recibir una carga nueva en un bucket de almacenamiento. En lugar de mantener un servidor encendido esperando archivos, se define una función que se activa automáticamente, procesa la imagen, guarda la versión redimensionada y termina su ejecución, sin consumir recursos mientras no esté activa.

Serverless destaca por su modelo de pago por uso, su escalabilidad automática y su bajo costo de mantenimiento para tareas event-driven. Es ideal para aplicaciones ligeras, flujos de trabajo por eventos, automatización de procesos y backends móviles o web simples. Sin embargo, no es

adecuado para procesos largos, estados persistentes o arquitecturas que requieran control detallado del entorno de ejecución.

Martin Fowler (2017) afirma que "serverless no es una panacea, pero es extremadamente poderoso cuando se aplica al tipo correcto de problema", especialmente para startups, prototipos y sistemas de automatización.

4.2.3 Factores clave en la selección de una arquitectura

Elegir una arquitectura adecuada depende de múltiples variables, entre ellas:

- Requisitos funcionales y no funcionales (rendimiento, seguridad, escalabilidad).
- Volumen de usuarios y transacciones esperadas.
- Equipo de desarrollo disponible (experiencia, tamaño, especialización).
- Mantenibilidad y evolución futura.
- Tiempo y presupuesto del proyecto.
- Tecnología y plataformas objetivo (web, móvil, nube, sistemas embebidos).

Un enfoque útil es utilizar vistas arquitectónicas (como las 4+1 de Kruchten) para analizar el sistema desde diferentes perspectivas: lógica, física, de procesos, de desarrollo y de casos de uso.

Documentación arquitectónica: Modelo de vistas 4+1

En proyectos de software de escala profesional, no basta con definir una arquitectura técnica; también es necesario comunicarla, analizarla y validarla desde múltiples perspectivas. Para ello, uno de los enfoques más utilizados en la industria es el modelo de vistas 4+1, propuesto por Philippe Kruchten en el contexto del desarrollo con UML y promovido por Rational Software.

Este modelo plantea que la arquitectura debe describirse mediante cinco vistas complementarias, cada una dirigida a un tipo diferente de interesado (stakeholder). El enfoque

ayuda a capturar tanto las decisiones estructurales como los flujos de operación, permitiendo verificar la coherencia del sistema frente a los requerimientos.

"Una arquitectura de software no puede describirse mediante un único modelo; el modelo 4+1 propone diferentes vistas para abordar las distintas preocupaciones de los stakeholders."
– Kruchten, P. (1995). Architectural Blueprints – The '4+1' View Model of Software Architecture.

Las cinco vistas del modelo 4+1

1. Vista lógica. Representa la estructura funcional del sistema desde el punto de vista del usuario o analista. Utiliza diagramas de clases, objetos y paquetes. Ejemplo: Modelar clases como Usuario, Pedido, Producto, y sus relaciones.
2. Vista de procesos. Describe la dinámica del sistema en tiempo de ejecución, incluyendo concurrencia, sincronización y comunicación entre procesos. Se usa para analizar rendimiento, escalabilidad y comportamiento concurrente. Ejemplo: Uso de hilos para procesar pagos mientras se genera la factura.
3. Vista de desarrollo (implementación). Presenta cómo se organiza el software desde la perspectiva del programador. Utiliza diagramas de componentes, carpetas, módulos y dependencias. Ejemplo: Separación en módulos como Gestión Usuarios, GestiónPedidos, APIExterna.
4. Vista física (despliegue). Representa la disposición del sistema en infraestructura real: servidores, nodos de red, contenedores. Muestra cómo se asignan los componentes del software al hardware. Ejemplo: El backend se despliega en un servidor Linux, la base de datos en otro nodo separado.
5. Vista de casos de uso (+1). Es la vista central que conecta todas las anteriores. Representa los requerimientos funcionales en forma de escenarios o

casos de uso, y sirve para validar que la arquitectura satisface lo esperado.
Ejemplo: Caso de uso "Realizar pedido", que recorre la vista lógica (clases), la de procesos (hilos), la de desarrollo (módulos), y la física (servidores).

¿Por qué usar este modelo?

- Mejora la comunicación con distintos actores del proyecto (desarrolladores, testers, clientes, DevOps).
- Facilita la detección de errores arquitectónicos al analizar el sistema desde varias ópticas.
- Refuerza la trazabilidad entre los requerimientos y la solución técnica.
- Promueve una documentación completa y estructurada del sistema.

Sommerville (2011) afirma que "documentar la arquitectura desde varias perspectivas proporciona una defensa robusta frente a cambios, ambigüedades o malentendidos durante el desarrollo" (p. 258).

4.3 Patrones de diseño

Los patrones de diseño son soluciones generales, probadas y reutilizables para problemas comunes que surgen durante la construcción de sistemas de software orientados a objetos. Aunque no son fragmentos de código específicos, sí representan estructuras de clases y objetos que pueden adaptarse a múltiples contextos, ayudando a mejorar la calidad del diseño, reducir errores recurrentes y acelerar el desarrollo.

"Un patrón de diseño describe un problema recurrente en el diseño de sistemas orientados a objetos, y una solución comprobada a ese problema que puede aplicarse en diversas situaciones."-*Gamma et al. (1994). Design Patterns: Elements of Reusable Object-Oriented Software*.

4.3.1 Introducción a los patrones de diseño

Los patrones no son recetas exactas, sino plantillas arquitectónicas que capturan buenas prácticas en situaciones

concretas. Se caracterizan por tener una estructura clara con tres partes principales:

- Problema: el contexto y la situación que genera la necesidad.
- Solución: la estructura propuesta (clases, roles, relaciones).
- Consecuencias: beneficios, limitaciones y consideraciones prácticas.

Estos patrones no resuelven problemas específicos del dominio del software (como facturación o seguridad), sino que abordan desafíos de diseño como instanciación, dependencia, comunicación, extensión, y mantenimiento.

4.3.2 Patrones de creación, estructurales y de comportamiento

Los patrones de diseño más conocidos fueron documentados por la "Banda de los Cuatro" (GoF) y se clasifican en tres grandes categorías:

Patrones creacionales

Singleton

El patrón Singleton garantiza que una clase tenga una única instancia en todo el sistema y proporciona un punto de acceso global a dicha instancia. Este patrón es útil cuando se necesita controlar el acceso centralizado a recursos compartidos, como archivos de configuración, conexiones de base de datos o controladores de sesión.

Estructura típica:

- Constructor privado
- Método estático que retorna la instancia única (getInstance())

Ejemplo práctico:

Una clase Configuración Sistema que carga y gestiona parámetros desde un archivo .json. Si múltiples clases del sistema accedieran y modificaran esta configuración por separado, podría generarse inconsistencia. Usar Singleton

asegura que todos los componentes trabajen sobre la misma instancia.

Ventajas:

- Control de acceso global
- Reducción de uso de memoria
- Facilita el logging, configuración y cache compartido

Factory Method

Este patrón define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Permite que el código cliente no dependa directamente de las clases concretas, lo que mejora la extensibilidad y la separación de responsabilidades.

Estructura:

- Clase base con método crear Producto().
- Subclases que implementan ese método para retornar instancias concretas.

Ejemplo:
Una interfaz Notificador tiene un método crear Notificador(). Subclases como Notificador Correo, NotificadorSMS y NotificadorPush implementan este método para retornar la instancia adecuada según el tipo de notificación requerida.

Beneficios:

- Aísla la lógica de instanciación
- Facilita la expansión del sistema con nuevos tipos sin modificar el código base

Builder

El patrón Builder separa la construcción de un objeto complejo de su representación final. Es útil cuando un objeto necesita ser creado paso a paso, especialmente si tiene múltiples configuraciones opcionales.

Estructura:

- Un Builder con métodos para establecer cada propiedad.

- Un Director que invoca los métodos del builder en orden.
- Un Producto que representa el objeto final.

Ejemplo**:**
La construcción de un objeto Reporte con secciones opcionales: portada, índice, gráficos, apéndices. El Builder permite armar distintas variantes del mismo tipo de reporte sin crear múltiples constructores.

Ventajas**:**

- Mejora la claridad del código
- Reduce errores en objetos con muchos parámetros
- Facilita la creación de versiones personalizadas

Patrones estructurales

Adapter

El patrón Adapter actúa como un traductor entre dos interfaces incompatibles. Permite que clases con estructuras distintas trabajen juntas sin necesidad de modificar su código original.

Estructura:

- Una clase Adapter implementa la interfaz esperada por el cliente y traduce las llamadas a la clase objetivo.

Ejemplo:
Un sistema espera una API con método obtener Datos(), pero una API externa solo ofrece fetchData(). El Adapter se encarga de hacer la conversión entre ambas.

Ventajas:

- Reutilización de código existente
- Integración de librerías o APIs externas sin alteración

Composite

El patrón Composite permite tratar objetos individuales y colecciones de objetos de la misma manera. Es ideal para estructuras jerárquicas donde los elementos pueden contener otros elementos del mismo tipo.

Estructura:

- Una interfaz Componente
    - Clases Hoja (elemento indivisible) y Compuesto (elemento que contiene otros componentes)

Ejemplo:
Una interfaz gráfica con elementos como Botón, Texto, y Formulario. El Formulario puede contener múltiples botones y campos, y todos implementan una misma interfaz para ser tratados uniformemente (ej. dibujar()).

Ventajas:

- Facilita la representación de árboles jerárquicos
- Unificación del tratamiento entre elementos simples y compuestos

Decorator

Decorator permite agregar funcionalidad extra a un objeto sin modificar su clase original. Se logra mediante el encapsulamiento del objeto en nuevas clases decoradoras que añaden comportamiento antes o después de invocar los métodos originales.

Estructura**:**

- Interfaz común
- Clase base
- Decoradores que implementan la interfaz y delegan en un objeto del mismo tipo

Ejemplo:
Un flujo de salida de texto puede ser decorado con Compresor, Encriptador, y Logger. Cada uno agrega una funcionalidad sin alterar la clase base de escritura.

Beneficios:

- Evita herencia excesiva
- Alta flexibilidad para combinaciones dinámicas

Patrones de comportamiento

Observer

Este patrón define una relación de dependencia uno-a-muchos entre objetos. Cuando el sujeto cambia de estado, notifica automáticamente a sus observadores.

Estructura:

- Sujeto con lista de observadores
- Observadores que implementan una interfaz actualizar()

Ejemplo:
En un sistema de monitoreo, cuando el sensor detecta una variación de temperatura, todos los componentes observadores (gráfico, alerta sonora, registro en base de datos) reciben la actualización en tiempo real.

Ventajas:

- Desacoplamiento entre productor y consumidor
- Comunicación automática entre componentes

Strategy

Strategy encapsula una familia de algoritmos intercambiables dentro de un mismo contexto. El algoritmo se selecciona en tiempo de ejecución y puede cambiar dinámicamente según la situación.

Estructura:

- Interfaz Estrategia
- Múltiples clases concretas que implementan ejecutar()
- Contexto que utiliza la estrategia

Ejemplo:
Un carrito de compras que selecciona una estrategia de descuento distinta: por cantidad, por cliente VIP, por fecha especial. Cambiar la estrategia no requiere modificar el carrito.

Ventajas:

- Elimina estructuras condicionales complejas
- Facilita la extensión de comportamientos

Command

Command encapsula una petición como un objeto, permitiendo parametrizar clientes con diferentes acciones, almacenar historiales o implementar deshacer/rehacer.

Estructura:

- Comando (ejecutar())
- Receptor (quien ejecuta la acción)
- Invocador (quien almacena y ejecuta comandos)

Ejemplo:
En una interfaz gráfica, los botones "Copiar", "Pegar" y "Deshacer" son comandos que encapsulan sus acciones. El sistema puede mantener un historial de comandos para gestionar acciones previas.

Ventajas:

- Separación entre emisor y ejecutor
- Soporte para colas de tareas, macros, historial

Tabla 8
Comparativa entre patrones de diseño

Patrón	Categoría	Propósito	Ejemplo práctico	Beneficios principales
Singleton	Creacional	Garantizar que solo exista una instancia de una clase	Clase ConfiguraciónSistema centralizada	Acceso controlado, ahorro de recursos
Factory Method	Creacional	Delegar la creación de objetos a subclases	Notificador → Correo, SMS, Push	Flexibilidad, separación de instanciación

Builder	Creacional	Construir objetos complejos paso a paso	Generación de Reporte con portada, índice, apéndices	Modularidad, legibilidad, personalización
Adapter	Estructural	Hacer compatibles interfaces distintas	Convertir fetchData() a obtenerDatos() en API externa	Reutilización, integración sin modificación
Composite	Estructural	Tratar objetos simples y compuestos de manera uniforme	Elementos UI como Botón, Formulario, Contenedor	Gestión jerárquica, simplicidad de tratamiento
Decorator	Estructural	Agregar funcionalidades a un objeto dinámicamente	Decoradores como Compresor, Encriptador, Logger	Extensibilidad, evita herencia rígida
Observer	Comportamiento	Notificar a múltiples objetos cuando cambia un sujeto	Sensor que actualiza Dashboard, Alarma, Log	Desacoplamiento, respuesta en tiempo real
Strategy	Comportamiento	Permitir cambiar el algoritmo en tiempo de ejecución	Estrategias de Descuento: por cantidad, VIP, por fecha	Eliminación de condicionales, alta flexibilidad

Command	Comportamiento	Encapsular acciones como objetos independientes	Botones Copiar, Pegar, Deshacer en una interfaz	Historia de acciones, macros, desacoplamiento

Nota. Elaboración propia

4.4 Tipos de diseño en ingeniería de software

El diseño de software se puede abordar desde distintos niveles y perspectivas, según el objetivo del sistema, el contexto tecnológico y los requerimientos específicos. En esta sección se describen tres enfoques clave: el diseño a nivel de componentes, el diseño basado en dominios específicos (DSL), y el diseño centrado en la experiencia del usuario (UX).

4.4.1 Diseño a nivel de componentes

El diseño a nivel de componentes se centra en definir cómo se organizarán y relacionarán los bloques funcionales independientes del sistema. Un componente es una unidad autónoma que ofrece servicios a través de interfaces bien definidas y puede ser desarrollada, reemplazada o desplegada de forma independiente.

Este tipo de diseño se alinea con arquitecturas como SOA (Service-Oriented Architecture) y microservicios, donde cada componente tiene responsabilidades claras y comunica mediante interfaces estándar (por ejemplo, APIs REST o eventos).

"El diseño basado en componentes promueve la reutilización, el mantenimiento independiente y la facilidad de prueba en entornos distribuidos."-*Szyperski, C. (2002). Component Software*.

Ejemplo práctico: en una aplicación de gestión académica, podrían existir componentes como Gestión de Estudiantes,

Gestión de Docentes, Gestión de Matrículas y Reportes. Cada uno expone operaciones específicas y puede evolucionar sin afectar a los demás.

4.4.2 Diseño basado en dominios específicos (DSL)

Este enfoque utiliza lenguajes específicos del dominio (Domain-Specific Languages) para expresar soluciones dentro de un contexto particular con mayor claridad y eficiencia que los lenguajes de propósito general. Los DSL pueden ser lenguajes visuales, declarativos o incluso conjuntos de anotaciones dentro del código.

"Un DSL permite capturar la lógica del dominio en una forma que sea comprensible tanto para expertos técnicos como para expertos del dominio."-*Fowler, M. (2010). Domain-Specific Languages*.

Ejemplos:

- Lenguajes como SQL (consultas sobre bases de datos), HTML/CSS (presentación web), VHDL (diseño de hardware), o lenguajes visuales para modelar BPMN en procesos de negocio.
- DSL internos, como funciones de alto nivel específicas en frameworks de testing (por ejemplo, describe, it, assert en Mocha o Jest).

Este tipo de diseño mejora la comunicación interdisciplinaria y permite construir herramientas especializadas para resolver problemas recurrentes en un nicho específico.

4.4.3 Diseño centrado en la experiencia del usuario (UX)

El diseño centrado en UX considera al usuario final como el eje principal del proceso de diseño. Este enfoque va más allá del diseño visual, incorporando aspectos de usabilidad, accesibilidad, emociones, contexto de uso y retroalimentación inmediata.

Según Norman (2013), "el buen diseño no solo debe ser funcional, también debe ser comprensible, predecible y emocionalmente satisfactorio para el usuario final".

Elementos clave del diseño UX:

- Personas: arquetipos de usuarios creados con base en datos reales.
- User journeys: recorrido que hace el usuario desde que entra al sistema hasta que logra su objetivo.
- Wireframes y prototipos interactivos: esquemas visuales que permiten validar antes de programar.
- Pruebas de usabilidad: evaluaciones con usuarios reales para detectar problemas antes del lanzamiento.

Ejemplo práctico: una aplicación móvil de banca digital debe permitir transferencias rápidas, mostrar saldos claramente y ofrecer retroalimentación inmediata. El diseño UX define cómo lograr esto desde el flujo, el lenguaje visual, hasta la interacción con errores.

4.5 Diseño de interfaces de usuario

El diseño de interfaces de usuario (UI) es una disciplina fundamental dentro del proceso de diseño de software, ya que establece el punto de contacto entre el sistema y las personas que lo utilizan. Una interfaz bien diseñada no solo mejora la usabilidad, sino que también reduce errores, acelera tareas, genera confianza y mejora la percepción global del sistema.

"El diseño de una buena interfaz de usuario es tanto una cuestión técnica como psicológica: requiere comprender las capacidades humanas y traducirlas en estructuras visuales claras y predecibles."-*Shneiderman et al. (2016). Designing the User Interface*.

4.5.1 Principios de diseño de interfaces

El diseño de interfaces efectivas se basa en principios universales de usabilidad y experiencia. Algunos de los más relevantes son:

- Visibilidad del estado del sistema. El sistema debe mantener informado al usuario sobre lo que está ocurriendo mediante retroalimentación inmediata.

Ejemplo: mostrar el progreso de carga o confirmaciones visuales tras una acción.

- Consistencia y estándares. El lenguaje, botones y flujos deben seguir patrones reconocibles. Por ejemplo, usar el ícono de lupa para búsqueda o mantener el botón "Guardar" en la misma ubicación.
- Control y libertad del usuario. El usuario debe poder corregir errores fácilmente o retroceder sin penalización. Funciones como "Deshacer", "Cancelar" o navegación hacia atrás son clave.
- Prevención de errores. Un buen diseño evita que ocurran errores antes que corregirlos. Ejemplo: deshabilitar el botón "Enviar" si un formulario no está completo.
- Reconocimiento antes que recuerdo. La interfaz debe mostrar opciones disponibles y no exigir al usuario que recuerde comandos. Menús contextuales o autocompletado son buenos ejemplos.
- Flexibilidad y eficiencia. Permitir a usuarios expertos realizar tareas de forma más rápida, sin entorpecer a los nuevos. Ejemplo: teclas rápidas, configuraciones avanzadas ocultas.

4.5.2 Evaluación y pruebas de usabilidad

Las pruebas de usabilidad permiten validar si la interfaz es funcional, intuitiva y efectiva antes de su despliegue definitivo. Existen distintos métodos:

- Pruebas con usuarios reales. Se observa cómo usuarios representativos interactúan con el sistema, detectando errores, bloqueos o comportamientos inesperados.
- Evaluación heurística. Expertos en usabilidad revisan la interfaz con base en principios reconocidos (como los 10 de Nielsen) para detectar inconsistencias o deficiencias.
- Análisis de tareas. Se descompone un flujo en pasos para evaluar si cada uno es necesario, lógico y eficiente.

Ejemplo: En una aplicación de registro médico, los usuarios reales (médicos y personal de salud) prueban la interfaz para agendar citas. Si se detecta que muchos no encuentran el botón "Confirmar", se reconsidera su ubicación o diseño.

4.5.3 Integración entre interfaz y lógica de negocio

El diseño de interfaz no debe desarrollarse de manera aislada, sino que debe estar íntimamente conectado con la lógica funcional del sistema. Un botón en la interfaz debe corresponder a una función válida en el backend; un campo de entrada debe validarse correctamente según las reglas del negocio.

Aquí es donde cobra valor la colaboración entre diseñadores de UX/UI y desarrolladores, garantizando que la experiencia visual tenga soporte técnico y viceversa.

Un patrón común que representa esta separación es MVC (Modelo-Vista-Controlador):

- Modelo: contiene la lógica y los datos del negocio.
- Vista: representa la interfaz visual que ve el usuario.
- Controlador: recibe las acciones del usuario y las traduce en operaciones sobre el modelo.

"Una buena interfaz no es solo bonita, sino una capa funcional alineada con el sistema, que transforma objetivos humanos en acciones computacionales exitosas."-Cooper, Reimann & Cronin (2014). About Face: The Essentials of Interaction Design.

Resumen del capítulo 4

Este capítulo abordó los fundamentos del diseño de software desde distintos niveles y enfoques. Se introdujeron los principios operativos del diseño, como la simplicidad, cohesión, abstracción y anticipación al cambio, y se profundizó en conceptos clave como la independencia funcional, la modularidad, el refinamiento progresivo y el ocultamiento de información. Posteriormente, se presentaron los estilos arquitectónicos más utilizados —como arquitectura en capas, microservicios y serverless— y el modelo de

documentación arquitectónica 4+1 de Kruchten. En la sección central del capítulo, se desarrollaron los patrones de diseño creacionales, estructurales y de comportamiento, explicando su utilidad en escenarios reales. También se analizaron los distintos tipos de diseño (a nivel de componentes, basado en dominios específicos y centrado en la experiencia del usuario) y se cerró con los principios del diseño de interfaces, resaltando la importancia de la usabilidad, las pruebas con usuarios y la integración con la lógica del negocio. Este conjunto de conocimientos proporciona al lector una visión integral, práctica y estructurada del diseño de software, fundamental para construir sistemas robustos, mantenibles y centrados en las personas.

Preguntas de revisión

1. ¿Cuáles son los principios operativos del diseño de software y cómo se aplican?
2. ¿Qué papel cumple la modularidad en la independencia funcional de un sistema?
3. ¿En qué consiste el modelo de vistas 4+1 de Kruchten y qué beneficio ofrece?
4. ¿Cuál es la diferencia entre los patrones Singleton, Factory y Builder?
5. ¿Cómo se diferencian los patrones estructurales de los patrones de comportamiento?
6. ¿Qué ventajas ofrece el diseño centrado en la experiencia del usuario?
7. ¿Qué técnicas existen para evaluar la usabilidad de una interfaz?

Autoevaluación

Parte A: Verdadero o Falso

Indique si las siguientes afirmaciones son verdaderas (V) o falsas (F).

1. La independencia funcional busca que cada módulo del software cumpla una tarea específica sin depender excesivamente de otros módulos. ( )

2. El refinamiento en el diseño de software consiste en descomponer un problema en niveles crecientes de abstracción para facilitar su comprensión e implementación. ( )
3. La modularidad del software implica dividir un sistema en componentes demasiado grandes, sin importar la cohesión interna ni el acoplamiento entre ellos. ( )
4. El ocultamiento de información promueve que los detalles internos de un módulo sean accesibles a todos los demás para evitar redundancias. ( )
5. En el diseño orientado a objetos, las clases encapsulan atributos y métodos que representan entidades del mundo real o conceptos abstractos. ( )
6. La arquitectura en capas organiza el software en diferentes niveles, donde cada capa ofrece servicios a la capa superior y utiliza los de la inferior. ( )
7. Los patrones de diseño son soluciones generales y reutilizables para problemas comunes en el desarrollo de software. ( )
8. El diseño de componentes se enfoca únicamente en la apariencia visual del software, sin considerar la interacción con otros elementos. ( )
9. El diseño de interfaces debe contemplar tanto la interacción humano-computadora como la comunicación entre módulos del sistema. ( )
10. El estudio de casos en diseño de software permite analizar aplicaciones reales o simuladas para comprender cómo se aplican los conceptos teóricos. ( )

Parte B: Selección múltiple

Seleccione la opción correcta para cada pregunta.

1. ¿Cuál de los siguientes principios busca reducir la dependencia entre módulos del sistema?
    a) Encapsulamiento
    b) Cohesión
    c) Bajo acoplamiento
    d) Refinamiento

2. ¿Qué vista del modelo 4+1 está destinada a representar los requerimientos funcionales?
    e) Vista lógica
    f) Vista de desarrollo
    g) Vista de casos de uso
    h) Vista de procesos

3. ¿Qué patrón permite cambiar dinámicamente el algoritmo utilizado sin modificar el contexto?

    i) Strategy
    j) Singleton
    k) Adapter
    l) Composite

4. ¿Cuál de los siguientes patrones encapsula una solicitud como un objeto independiente?

    m) Observer
    n) Builder
    o) Command
    p) Factory

5. ¿Qué enfoque permite representar soluciones específicas mediante mini-lenguajes especializados?

    q) Arquitectura orientada a eventos
    r) Diseño centrado en el usuario
    s) Diseño basado en componentes
    t) Diseño basado en dominios específicos (DSL)

Parte C: Relacione las columnas

Relaciona cada término del grupo A con su definición correspondiente del grupo B.

Patrón	**Función**
a) Observer	( ) Crear objetos paso a paso
b) Builder	( ) Notificar automáticamente a múltiples objetos
c) Decorator	( ) Agregar funcionalidades sin modificar la clase base
d) Adapter	( ) Hacer compatibles interfaces incompatibles

Ejercicio de aplicación final

"Sistema web para reservas de espacios deportivos"

El Municipio Deportivo de Ciudad Verde gestiona canchas (fútbol, básquet, tenis) en múltiples complejos. Hoy las reservas se hacen por teléfono o en ventanilla, lo que genera sobrecupos y quejas por falta de confirmaciones. Se encarga el desarrollo de un sistema web para: registro de usuarios, consulta de horarios y disponibilidad en tiempo real, reserva y pago (si aplica), notificaciones (email/SMS) y evaluación del servicio. Se pide una arquitectura escalable (capaz de crecer a nuevos complejos) y un diseño centrado en el usuario para minimizar errores de reserva.

Actores y roles

- Usuario: se registra, consulta disponibilidad, reserva/cancela, evalúa el servicio.
- Administrador del complejo: crea espacios, define horarios/tarifas, bloquea canchas por mantenimiento, gestiona incidencias.
- Pasarela de pago (opcional): procesa cobros y devuelve estado de transacción.

- Servicio de notificaciones: envía confirmaciones, recordatorios y avisos de cambios.

Alcance (versión inicial)

- Búsqueda por complejo/fecha/deporte.
- Reserva/cancelación con reglas simples (p. ej., máx. 2 horas por día).
- Confirmación y recordatorios por correo/SMS.
- Panel de administración para gestión de espacios, horarios, bloqueos y aforo.
- Registro de evaluaciones (1–5) y comentarios post-uso.

Supuestos y restricciones

- Integración con una pasarela de pago.
- Cumplimiento de protección de datos personales.
- Acceso web desde escritorio y móvil (responsivo).

Objetivos del ejercicio

- Aplicar conceptos de diseño (abstracción, modularidad, ocultamiento de información, independencia funcional).
- Definir una arquitectura adecuada (capas o microservicios) justificando decisiones.
- Seleccionar y ejemplificar patrones de diseño pertinentes al dominio.
- Modelar el sistema con UML (clases; separación MVC).
- Diseñar un prototipo de interfaz centrado en el usuario.
- Incorporar principios de UX que reduzcan errores y mejoren la experiencia.

Actividades para desarrollar

1. Arquitectura general del sistema (capas o microservicios)

- Propón el estilo (p. ej., en capas: presentación, aplicación/servicios, dominio, infraestructura; o microservicios: reservas, catálogo de espacios, notificaciones, pagos).
- Justifica con 3–5 criterios (escalabilidad, despliegue, equipo, complejidad operativa, mantenibilidad).

2. Patrones de diseño (al menos 3) con ejemplos concretos

- Sugeridos (elige y justifica):
- Observer para notificaciones (reserva confirmada/cancelada → avisos a correo/SMS).
- Strategy para cálculo de tarifas (hora pico, tipo de usuario, promociones).
- Factory/Abstract Factory para construir objetos de Reserva según tipo de espacio.
- Adapter para integrar distintas pasarelas de pago hoy/mañana.
- Singleton para gestor central de configuración (con cautela).
    a. Para cada patrón: problema → solución del patrón → mini-ejemplo en el dominio.

3. Prototipo de interfaz del flujo de reserva (papel o digital)

- Pantallas mínimas: catálogo/selector, detalle del espacio y horario, confirmación, feedback de resultado.
- Indica mensajes de error/éxito y estados de carga.

4. Principios de diseño UX (al menos 5) aplicados

- Sugeridos: visibilidad del estado del sistema, prevención y recuperación de errores (confirmaciones, deshacer), consistencia y estándares, retroalimentación inmediata, control y libertad del usuario, accesibilidad (contraste, etiquetas, foco), minimizar carga cognitiva (pasos guiados).

Referencias Bibliográficas

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … Thomas, D. (2001). Manifesto for agile software development. Agile Alliance. https://agilemanifesto.org

Boehm, B. W. (1981). Software engineering economics. Prentice Hall.

Boehm, B. W. (1988). A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes, 11(4), 14–24. https://doi.org/10.1145/12944.12948

DeMarco, T., & Lister, T. (1987). Peopleware: Productive projects and teams. Dorset House Publishing.

McConnell, S. (2006). Software estimation: Demystifying the black art. Microsoft Press.

Nielsen, J. (1994). Usability engineering. Morgan Kaufmann.

Norman, D. A. (2013). The design of everyday things (Revised and expanded ed.). Basic Books.

Pressman, R. S., & Maxim, B. R. (2021). Software engineering: A practitioner's approach (9th ed.). McGraw-Hill Education.

Sommerville, I. (2011). Software engineering (9th ed.). Addison-Wesley.

Sommerville, I. (2016). Software engineering (10th ed.). Pearson.

Wake, W. (2001). The INVEST acronym. XP123. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

John Patricio Orellana Preciado, Mgs.
Ingeniero de Sistemas por la Universidad Técnica de Machala, con estudios de maestría en Gerencia y Docencia en Educación Superior por la Universidad de Guayaquil, y en Tecnologías de la Información por la Universidad Católica de Cuenca. Desde el año 2014 se desempeña como docente en la Universidad Técnica de Machala, donde ha impartido diversas asignaturas en el área de Tecnologías de la Información, especialmente relacionadas con la Ingeniería de Software. Sus líneas de investigación incluyen la ingeniería de software y la visión artificial, con publicaciones científicas destacadas en temas como reconocimiento de placas vehiculares, análisis de seguridad en redes frente a ataques VLAN Hopping, y estudios comparativos sobre herramientas de inteligencia artificial aplicadas al desarrollo web. Además, ha participado activamente en procesos de acreditación institucional, colaborando en la evaluación y mejora de las carreras de Medicina y Enfermería.
https://orcid.org/0000-0001-7482-1440
jporellana@utmachala.edu.ec

ISBN: 978-9942-53-116-2

Compás
capacitación e investigación