FUNDAMENTOS

del Diseño Orientado a Objetos en Ingeniería de Software:

Desarrollo en Java
como base para la formación investigativa

Miguel Ángel Fernández Marín
Débora González Tolmo

FUNDAMENTOS

del Diseño Orientado a Objetos en Ingeniería de Software:

Desarrollo en Java
como base para la formación investigativa

Miguel Ángel Fernández Marín
Débora González Tolmo

Dirección editorial: PhD. Jorge Luis León-González
Representante del sello editorial: Mg. Carmen Priscilla Guerra-Maldonado
Diseño de carátula y maquetación: D.I. Yunisley Bruno-Díaz

ISBN: 978-9942-7448-3-8

La evaluación científica y metodológica de la obra se realizó a partir del método de Revisión por Pares Abierta (Open Peer Review).

Editorial UMET
Universidad Metropolitana
Gral. Francisco Robles 411, Quito,
Ecuador, 170143

ÍNDICE

COMITÉ
EDITORIAL

PREFACIO

La Ingeniería de Software ha evolucionado significativamente en las últimas décadas, pasando de enfoques puramente estructurados a paradigmas más flexibles que permiten enfrentar la creciente complejidad de los sistemas modernos. Dentro de este panorama, la Programación Orientada a Objetos (POO) se ha consolidado como un pilar fundamental del desarrollo de software. Su enfoque basado en clases, objetos, herencia y encapsulación proporciona no solo una manera de organizar el código, sino también un marco conceptual que permite modelar problemas del mundo real de manera estructurada y coherente.

Este libro, Fundamentos del Diseño Orientado a Objetos en Ingeniería de Software: Desarrollo en Java como base para la formación investigativa, surge de la necesidad de ofrecer a estudiantes y profesionales una guía integral que combine la teoría, la práctica y la reflexión crítica. Nuestro objetivo principal es que el lector comprenda no solo la sintaxis y el funcionamiento de un lenguaje de programación como Java, sino que también desarrolle habilidades de análisis, diseño y evaluación que le permitan construir software robusto, mantenible y escalable.

A lo largo de las páginas que siguen, se exploran los conceptos fundamentales de la POO: desde la definición de clases y objetos, hasta principios avanzados como polimorfismo, herencia múltiple, genericidad y relaciones entre entidades. Se profundiza también en la aplicación de los principios SOLID, la cohesión y el acoplamiento, y la refactorización de código como herramientas esenciales para garantizar que los sistemas puedan evolucionar sin comprometer su integridad. Cada capítulo ha sido diseñado para ofrecer explicaciones claras acompañadas de ejemplos prácticos, diagramas UML y ejercicios integradores, de modo que el lector pueda experimentar de manera activa y directa con los conceptos abordados.

Un aspecto central de este libro es su enfoque en el diseño extensible y robusto. Comprender cómo anticipar cambios futuros, encapsular lo que varía y estructurar correctamente las clases y responsabilidades es esencial para evitar errores comunes, como la aparición de clases “Dios” o la proliferación de condicionales que dificultan la mantenibilidad. Además, se dedica un espacio considerable al manejo de errores y validaciones, mostrando cómo diseñar sistemas que no solo funcionen correctamente ante entradas esperadas, sino que sean capaces de manejar situaciones inesperadas sin comprometer la estabilidad del software. Este enfoque fortalece la resiliencia de los sistemas y forma parte de las buenas prácticas que todo ingeniero de software debe dominar.

Otro objetivo fundamental de esta obra es fomentar la formación investigativa del lector. Se promueve la reflexión crítica mediante estudios de caso, análisis metodológicos y la comparación de alternativas de diseño. De esta manera, el aprendizaje va más allá de memorizar conceptos; el estudiante aprende a tomar decisiones fundamentadas, evaluar sus impactos y justificar sus elecciones de diseño. Este enfoque prepara a los profesionales para enfrentarse a desafíos reales en entornos académicos y laborales, donde la toma de decisiones correcta y basada en criterios sólidos es crucial.

Asimismo, el libro enfatiza la importancia de los patrones de diseño como herramientas que permiten solucionar problemas recurrentes de manera estructurada. Estrategias como el patrón Strategy, Observer, Factory Method, Decorator y Singleton se presentan con ejemplos claros, mostrando cómo su correcta implementación puede simplificar la evolución de un sistema, mejorar su legibilidad y reducir errores. Esta sección es especialmente relevante para quienes buscan desarrollar software que pueda crecer sin comprometer su calidad ni su coherencia interna.

Finalmente, este libro refleja nuestra convicción de que el aprendizaje de la POO debe ser integral. No se trata únicamente de aprender un lenguaje o ejecutar instrucciones; el objetivo es formar ingenieros de software capaces de pensar, analizar y crear soluciones que sean eficientes, seguras y sostenibles. Esperamos que esta obra sirva como una guía confiable tanto para docentes como para estudiantes y profesionales, y que inspire un enfoque disciplinado y creativo hacia el diseño orientado a objetos.

En esta obra invitamos al lector a iniciar un recorrido que combina teoría, práctica y reflexión, con el propósito de fortalecer sus competencias técnicas y analíticas, y de fomentar una visión crítica y profesional del desarrollo de software orientado a objetos. La intención es que, al finalizar la lectura, el lector no solo maneje los conceptos fundamentales, sino que también haya desarrollado una capacidad investigativa y de resolución de problemas que le permita enfrentar los retos del software contemporáneo con confianza y eficiencia.

Los Autores

INTRODUCCIÓN

La formación en programación orientada a objetos y diseño de software constituye hoy un eje estructural en la educación superior en el ámbito de las ciencias informáticas. No se trata únicamente de aprender a escribir código funcional, sino de desarrollar la capacidad de pensar, modelar y construir sistemas que respondan a problemas complejos, evolucionen en el tiempo y mantengan niveles adecuados de calidad. En este sentido, la programación orientada a objetos se consolida como un paradigma que articula abstracción, modularidad y reutilización, permitiendo al estudiante aproximarse al desarrollo de software de manera sistemática y profesional.

Este libro surge como un texto base de estudio diseñado para acompañar el proceso formativo en asignaturas de programación de nivel universitario, particularmente en aquellas donde se introducen y profundizan los fundamentos de la programación orientada a objetos, el diseño de clases, las relaciones entre objetos y los principios de buen diseño. Su contenido ha sido estructurado para servir tanto como material de apoyo en el aula como para el estudio autónomo, proporcionando explicaciones conceptuales claras, ejemplos progresivos y análisis metodológicos que permiten comprender no solo el cómo, sino también el porqué de cada decisión de diseño.

En las carreras de Ingeniería en Sistemas, Tecnología en Big Data e Inteligencia de Negocios y Tecnología en Desarrollo de Software, los estudiantes suelen enfrentarse tempranamente a múltiples lenguajes, herramientas y entornos tecnológicos. Sin embargo, la experiencia académica demuestra que muchas de las dificultades posteriores no provienen del desconocimiento de la sintaxis, sino de una comprensión insuficiente de los principios de diseño y de la correcta estructuración del software. Este libro responde a esa necesidad formativa, integrando contenidos

que habitualmente se abordan de forma fragmentada y ofreciendo una visión coherente y acumulativa del desarrollo orientado a objetos.

Desde una perspectiva curricular, el texto se concibe como un material articulador entre las asignaturas de programación básica, programación orientada a objetos, diseño de software y asignaturas aplicadas, permitiendo al estudiante consolidar conocimientos que serán reutilizados en distintos momentos de su trayectoria académica. Conceptos como cohesión, acoplamiento, relaciones entre clases, uso adecuado de herencia, composición, interfaces y patrones de diseño son abordados con profundidad, reconociendo su papel central en la construcción de soluciones reales y sostenibles.

Un elemento distintivo de esta obra es su énfasis en el razonamiento metodológico. Cada capítulo no se limita a presentar soluciones correctas, sino que analiza alternativas incorrectas o ingenuas, explicando sus limitaciones y consecuencias. Este enfoque pedagógico busca desarrollar en el estudiante un criterio de diseño sólido, indispensable tanto para el desarrollo profesional como para la participación en proyectos académicos avanzados. Aprender a justificar una decisión técnica es tan importante como aprender a implementarla.

El enfoque del texto no se limita a la enseñanza tradicional de programación, sino que responde a una necesidad institucional más amplia: preparar a los estudiantes para participar de manera efectiva en proyectos de investigación aplicada donde la calidad del diseño, la claridad del modelo y la correcta gestión de errores son requisitos fundamentales. En particular, este libro se vincula directamente con el proyecto institucional de I+D+i titulado “Inteligencia Artificial y Aprendizaje Profundo para la Transformación Digital Universitaria y la Clasificación de Péptidos Antimicrobianos en Bioinformática”, desarrollado en el marco de la Línea 3: Transformación Digital y Audiovisual y del Programa 5: Transformación Digital en la Educación Superior. El proyecto

demanda sistemas bien estructurados para el manejo de datos, la implementación de modelos y la integración de componentes de software complejos. Además, involucra a docentes y estudiantes de las carreras mencionadas y demanda competencias sólidas en diseño de software, gestión de errores, estructuración por capas y pensamiento sistemático.

La participación estudiantil en proyectos de inteligencia artificial, Big Data y bioinformática requiere una base sólida en programación orientada a objetos que garantice la reproducibilidad, mantenibilidad y escalabilidad de las soluciones desarrolladas. Los principios y técnicas abordados en este libro, desde el modelado UML hasta la gestión profesional de validaciones y excepciones, constituyen competencias transversales que permiten a los estudiantes desenvolverse con mayor solvencia en entornos de investigación multidisciplinarios y tecnológicamente exigentes.

Adicionalmente, el texto promueve una visión del software como producto intelectual y científico, no solo como artefacto funcional. Esta perspectiva resulta clave para la formación de estudiantes que aspiran a continuar estudios de posgrado, integrarse en grupos de investigación o participar en proyectos de innovación tecnológica. El rigor conceptual, la claridad estructural y la disciplina en el diseño que se fomentan a lo largo del libro reflejan estándares propios del desarrollo de software académico y científico.

Finalmente, este libro se plantea como un instrumento de referencia permanente a lo largo de la carrera. Su lectura no se agota en una asignatura específica, sino que acompaña al estudiante en diferentes etapas de su formación, sirviendo como guía para la toma de decisiones de diseño, la revisión de código y la comprensión de sistemas más complejos. De este modo, el texto contribuye a fortalecer una cultura de calidad, reflexión y mejora continua en el desarrollo de software, alineada con las necesidades académicas, profesionales e investigativas de la universidad.

01

Conceptos fundamentales de la Programación Orientada a Objetos

1.1. Principios fundamentales de la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) constituye uno de los paradigmas de programación más influyentes y vigentes en el desarrollo de software. Su importancia radica en que facilita la modularidad, reutilización de código y escalabilidad de los sistemas, lo que resulta indispensable en un entorno donde los proyectos crecen en complejidad y se requiere mantener estándares de calidad elevados. De acuerdo con Sommerville (2020), la POO organiza el software en unidades llamadas objetos, encapsulando tanto datos como comportamientos, lo cual permite estructurar los sistemas de forma más cercana al mundo real.

Este enfoque ha trascendido el ámbito académico y se ha convertido en un estándar en la industria tecnológica, aplicándose en áreas tan diversas como sistemas empresariales, inteligencia artificial, desarrollo de videojuegos y aplicaciones móviles.

desarrollo de videojuegos y aplicaciones móviles.

Los principios de la programación orientada a objetos incluyendo abstracción, encapsulamiento, herencia y polimorfismo son característicos de lenguajes modernos y apoyan la modularidad y reutilización en software complejo (Chaudhary et al., 2025). A continuación, se definen estos conceptos.

La **abstracción** permite modelar entidades del mundo real mediante clases y objetos, representando únicamente las características esenciales y ocultando los detalles irrelevantes para el contexto del problema, lo que facilita la modularidad y la reutilización del software (Sánchez Durán, 2026).

Ejemplo: representar a un *Usuario* con atributos básicos (nombre, correo, edad), sin necesidad de incluir toda su información personal.

El **encapsulamiento** Parmar y Parmar (2024) lo describen como uno de los conceptos fundamentales de POO y se refiere al agrupamiento de datos y comportamiento en una unidad lógica, protegiendo los datos internos mediante interfaces bien definidas. Esto quiere decir que los datos internos de un objeto restringen su acceso, lo que asegura consistencia y seguridad en las operaciones.

La **herencia** en la programación orientada a objetos permite que una clase derive atributos y métodos de otra, facilitando así la reutilización de código y la extensión de funcionalidades sin duplicación (Ghorpade y Wadkar, 2024; Parmar y Parmar, 2024).

El **polimorfismo** en la programación orientada a objetos permite que distintos objetos respondan de manera única al mismo mensaje o llamada de método, lo que incrementa la flexibilidad, extensibilidad y reutilización del diseño del software (Ghorpade y Wadkar, 2024).

Estudios recientes reafirman que la programación orientada a objetos (POO) no es solo un modelo de programación, sino un paradigma con impacto significativo en la calidad del software (Heričko y Šumak, 2023; Magableh et al., 2024)software systems undergo continuous correction and enhancement activities due to emerging faults, changing environments, and evolving requirements, making this phase expensive and time-consuming, often exceeding the initial development costs. To understand and manage software under development and maintenance better, several maintainability measures have been proposed. The Maintainability Index is commonly used as a quantitative measure of the relative ease of software maintenance. There are several Index variants that differ in the factors affecting maintainability (e.g., code complexity, software size, documentation. En particular, la relación entre métricas de diseño

orientado a objetos y mantenibilidad ha sido explorada empíricamente en múltiples sistemas Java, mostrando que variaciones en métricas como acoplamiento, complejidad ciclomática, líneas de código e índices de mantenibilidad reflejan diferencias reales en la facilidad de mantenimiento. Además, se han propuesto métricas específicas de POO que evalúan características como la encapsulación, la herencia, el polimorfismo y los patrones de diseño, que contribuyen a modularidad, reutilización y extensibilidad del software. Estas investigaciones consolidan la idea de que en POO los principios de diseño, más que reglas específicas, funcionan como estándares implícitos que promueven calidad y mantenibilidad del código.

Beneficios de la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) ofrece beneficios fundamentales para el desarrollo de software moderno, facilitando tanto la construcción inicial como la evolución de sistemas complejos. Un beneficio clave es la modularidad, que permite estructurar el software en unidades independientes (objetos y clases) que pueden desarrollarse y probarse por separado, reduciendo la complejidad del sistema y mejorando la trazabilidad de cambios. Este enfoque modular facilita la gestión de sistemas de gran escala y la incorporación de nuevas funcionalidades sin comprometer partes previas del sistema (Vera y Vera, 2023).

La aplicación de los principios de la programación orientada a objetos resulta especialmente relevante en el desarrollo de software educativo, donde la modularidad, la reutilización y la claridad del contextos formativos (Fernández Marín et al., 2019).

Adicionalmente, la POO potencia la reutilización de código, ya que mediante mecanismos como la herencia y la composición es posible aprovechar

implementaciones existentes en nuevos contextos, reduciendo la duplicación y los errores asociados a código repetido. Este beneficio se traduce en mayor eficiencia del desarrollo y menor tiempo de implementación de nuevas funcionalidades (Chen y Huang, 2025).

La escalabilidad es otro aspecto favorecido por la POO, pues las estructuras jerárquicas y los patrones de diseño basados en objetos permiten extender aplicaciones de forma ordenada conforme crecen los requisitos del dominio. La organización del software en objetos autónomos facilita adaptar sistemas a nuevas demandas sin reescribir componentes básicos (Vera y Vera, 2023).

Implementación en Java

Java es reconocido como un lenguaje que implementa de forma consistente y completa los principios de la programación orientada a objetos, facilitando la construcción de sistemas modulares, reutilizables y mantenibles. Según un estudio exhaustivo sobre el desarrollo orientado a objetos, los lenguajes modernos basados en OOP (como Java) están fundamentados en los pilares clásicos de la abstracción, encapsulación, herencia y polimorfismo, que juntos permiten el diseño de software robusto y flexible (Chaw, 2024).

En Java, la definición de clases y objetos constituye el núcleo del paradigma; una clase actúa como prototipo de un tipo de objeto, encapsulando tanto datos como comportamientos relacionados en una sola unidad. Esta organización de código permite representar entidades del mundo real de forma directa dentro del programa, mejorando la comprensión y la mantenibilidad del software (Parmar y Parmar, 2024).

El encapsulamiento, implementado mediante modificadores de acceso como private, protected

y public, protege los datos internos de un objeto al restringir su acceso desde fuera de la clase, promoviendo la ocultación de información y la integridad del diseño. Esta característica es fundamental para prevenir dependencias innecesarias y reducir errores comunes de programación.

La herencia permite que una clase derive atributos y métodos de otra, promoviendo la reutilización de código y la extensión controlada de comportamientos comunes. Este mecanismo no solo reduce la duplicación de código, sino que también facilita la creación de jerarquías lógicas que reflejan relaciones semánticas entre entidades software.

Finalmente, el polimorfismo permite que un mismo método o interfaz se comporte de diferentes maneras según el tipo de objeto que lo invoque, aumentando significativamente la flexibilidad del diseño y la capacidad de adaptación del software ante cambios en los requisitos. Este principio contribuye a que los sistemas orientados a objetos puedan responder de forma coherente a mensajes comunes enviados a diferentes objetos, lo cual es un elemento clave en la extensión y generalización de comportamientos.

Ejemplo básico:

```
public class Producto {
    private String nombre;
    private double precio;
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
```

```
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Precio: "
+ precio);
    }
}
```


1.2. Principios SOLID y Buenas Prácticas de Diseño en la Programación Orientada a Objetos

Uno de los aportes más significativos al desarrollo orientado a objetos en las últimas décadas son los principios SOLID, propuestos por Robert C. Martin y consolidados como guía universal de buenas prácticas de diseño. Estos principios buscan garantizar que el código sea fácil de mantener, flexible ante cambios, reutilizable y robusto frente a nuevas funcionalidades. Su estudio resulta imprescindible en la formación universitaria, pues prepara al estudiante para afrontar proyectos reales en los que el crecimiento de requisitos y la colaboración en equipos de desarrollo es la norma. Diversos estudios recientes confirman que la aplicación de los principios SOLID favorece la comprensión del código y la mantenibilidad en proyectos complejos (Cabral et al., 2024).

Los cinco principios se definen de la siguiente manera:

Single Responsibility Principle (SRP) – Principio de Responsabilidad Única

Single Responsibility Principle (SRP), el Principio de Responsabilidad Única, establece que cada clase o módulo debe tener una única responsabilidad y, por tanto, una sola razón para cambiar, lo que contribuye a la claridad, cohesión y mantenibilidad del software. Según investigaciones recientes sobre diseño de software orientado a objetos, la aplicación de los principios SOLID,

incluido el SRP, mejora la comprensión del código, reduce el acoplamiento y favorece la extensibilidad de sistemas complejos (Cabral et al., 2024). Un error común en estudiantes es diseñar clases monolíticas que agrupan demasiadas funcionalidades, dificultando la mantenibilidad y el cumplimiento del principio de responsabilidad única. Por ejemplo, una clase GestorUsuarios que además de registrar usuarios también gestione la base de datos y envíe notificaciones. Este diseño genera acoplamiento innecesario y dificulta el mantenimiento.

Ejemplo incorrecto:

```
public class GestorUsuarios {
    public void registrarUsuario(String nombre) {
        // Lógica de registro
    }
    public void guardarEnBaseDatos(String nombre) {
        // Lógica de persistencia
    }
    public void enviarCorreo(String nombre) {
        // Lógica de notificación
    }
}
```

La clase GestorUsuarios concentra varias responsabilidades:

- Registrar al usuario.
- Guardar directamente en la base de datos.
- Enviar correos de notificación.

Esto la convierte en una clase monolítica o "clase Dios". El problema es que cualquier cambio en la base de datos o en la forma de notificar obliga a modificar esta misma clase, aumentando el acoplamiento y reduciendo la mantenibilidad.

Ejemplo corregido aplicando SRP:

```
public class GestorUsuarios {
    private RepositorioUsuarios repositorio;
    private ServicioNotificacion notificador;

    public void registrarUsuario(String nombre) {
        repositorio.guardar(nombre);
        notificador.enviarCorreo(nombre);
    }
}
```

La clase GestorUsuarios se encarga solo del proceso de registro, delegando otras tareas a clases especializadas:

- RepositorioUsuarios gestiona la persistencia de datos.
- ServicioNotificacion se encarga de enviar correos.

Así cada clase tiene un único motivo de cambio y se respeta el Principio de Responsabilidad Única (SRP). El diseño es más limpio, mantenible y fácil de extender en proyectos reales.

Open/Closed Principle (OCP) – Principio Abierto/Cerrado

Establece que los módulos de software deben estar abiertos a la extensión, pero cerrados a la modificación,

de modo que nuevas funcionalidades puedan incorporarse sin alterar el código previamente validado. Este principio es ampliamente reconocido como un pilar del diseño orientado a objetos, ya que reduce el riesgo de errores, mejora la mantenibilidad y favorece la evolución controlada de sistemas complejos (Meyer, 1994; Cabral et al., 2024; Sommerville, 2020).

Esto significa que una clase debe permitir añadir nuevas funcionalidades sin necesidad de modificar su código original, reduciendo el riesgo de introducir errores en sistemas ya probados.

Ejemplo práctico:

Supongamos que tenemos una clase Figura con un método calcularArea(). En lugar de modificar la clase cada vez que aparece una nueva figura geométrica, se definen subclases que implementen el cálculo específico.

```
abstract class Figura {
    public abstract double calcularArea();
}

class Circulo extends Figura {
    private double radio;
    public Circulo(double radio) { this.radio = radio; }
    @Override
     public double calcularArea() { return Math.PI * radio * radio; }
}
```

```
class Rectangulo extends Figura {
    private double base, altura;
    public Rectangulo(double base, double altura) {
        this.base = base; this.altura = altura;
    }
    @Override
    public double calcularArea() { return base * altura; }
}
```


1.3. El lenguaje unificado de modelado

El lenguaje unificado de modelado (UML) fue adoptado por los miembros del Object Managemente Group (OMG) como estándar en 1997. El mismo constituye un lenguaje de modelado de propósito general que pueden usar todos los modeladores. No tiene propietario y está basado en el común acuerdo de gran parte de la comunidad informática (Rumbaugh et al., 2006).

Es un lenguaje muy utilizado en el desarrollo de software orientado a objetos, para el diseño de diagramas y modelos, el cual facilita la documentación, diseño y representación de un software. Establece estereotipos claros y entendibles en su conformación que permite ser entendible para la comunidad de desarrollo y para personas no involucradas con esta formación, facilitando la comunicación de un equipo multidisciplinar a través de representaciones visuales de la estructura, comportamiento e interacción de los componentes del software antes de su implementación. Según Ramírez Jiménez et al. (2024)adquirir una mayor habilidad para plantear y representar la solución de un proyecto o un problema de cualquier índole, ya sea informático, empresarial, industrial, educativo, entre

otros, mediante diferentes tipos de diagramas que puedan ser interpretados por cualquier persona aun cuando no esté familiarizado con los aspectos técnicos de la computación.\nUML es un estándar de modelado visual e internacional utilizado por los desarrolladores de sistemas de software para mantener una comunicación constante y efectiva con los actores involucrados (analistas, desarrolladores o usuarios finales constituye una poderosa herramienta que aporta importantes beneficios a los estudiantes de educación superior de forma general, con la finalidad de que desarrollen un pensamiento computacional, adquieran una mayor habilidad para plantear y representar la solución de un proyecto o un problema de cualquier índole.

Por eso, conocer este lenguaje para los profesionales de software específicamente, en cualquiera de sus roles es de vital importancia, pues con su uso se logra una mejor planificación del software ya que previo a la codificación permite elaborar un diseño estereotipado de la arquitectura facilitando su discusión a través de diagramas bien estructurados. Esto provoca que internamente en el equipo mejore la comunicación de sus participantes, al igual que externamente entre equipos ya que se utiliza un lenguaje fácil y común para desarrolladores, diseñadores, clientes y probadores. Además, con este enfoque organizado ayuda a detectar problemas discutibles por el equipo en la fase de diseño previo a escribir el código, reduciendo así errores en el desarrollo. También impacta sobre la documentación y mantenimiento, debido a que posee mecanismos sólidos para lograr estos propósitos ayudando a prever mejoras y correcciones en el futuro. Al mismo tiempo, favorece la reutilización de código a través de un diseño entendible, donde los especialistas pueden visualizar los componentes y sus comunicaciones, facilitando la modularidad y así ser reutilizables estas fracciones de código en diferentes proyectos. Además, suele ser

compatible con metodologías ágiles y tradicionales como SCRUM, modelos en cascada o Rational Unified Process por sus siglas en inglés RUP.

Un ejemplo de su uso y acoplamiento con metodologías híbridas utilizadas es en el trabajo (Fernández Marín y González Tolmo, 2020)educational multimedia, which provide the necessary information for the student to study in an enjoyable way, fix the knowledge and correct any mistakes that may be made in the process. An Application Development Methodology, which considering the requirements, design and development of an educational project. Complementing it with the UnifiedSoftwareDevelopmentProcessanddocumenting it with the help of UML (Modeling Language donde los autores muestran una integración entre RUP y una metodología de desarrollo de aplicaciones multimedia interactiva, teniendo en cuenta los modelos a través de UML.

Aunque UML tiene múltiples diagramas organizados en categorías, este libro se referirá a un diagrama en particular que es el diagrama de clases, el cual para Ramírez Jiménez et al. (2024)adquirir una mayor habilidad para plantear y representar la solución de un proyecto o un problema de cualquier índole, ya sea informático, empresarial, industrial, educativo, entre otros, mediante diferentes tipos de diagramas que puedan ser interpretados por cualquier persona aun cuando no esté familiarizado con los aspectos técnicos de la computación.\nUML es un estándar de modelado visual e internacional utilizado por los desarrolladores de sistemas de software para mantener una comunicación constante y efectiva con los actores involucrados (analistas, desarrolladores o usuarios finales está entre los diagramas más comunes pues representa los componentes fundamentales que dan forma a la estructura de clases. Por eso

constituye un tipo de diagrama de estructura, que facilita los estereotipos necesarios para representar las clases con sus atributos, métodos y relaciones como asociación, agregación, composición y herencia. Los diferentes diagramas de clases serán representados por una herramienta en particular en este caso Enterprise Architect (EA), que es muy utilizada para el modelado de software y arquitectura de sistemas, proporcionando una plataforma robusta para diseñar y documentar aplicaciones complejas.

Elementos UML claves en un diagrama de clases

Las clases representan los objetos del sistema, su representación son rectángulos con tres secciones. Implementa modificadores de visibilidad privado (-), protegido (#) y publico (+). Cada uno de ellos con un propósito claro, por ejemplo, un atributo o método con visibilidad privada solo se puede acceder a él dentro de la clase, si fueran protegidos significa que es accesible dentro de la clase y sus subclases y si es público se puede acceder a estos desde cualquier parte (Figura 1.1).

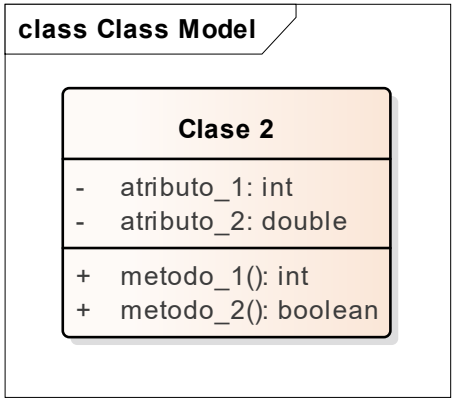
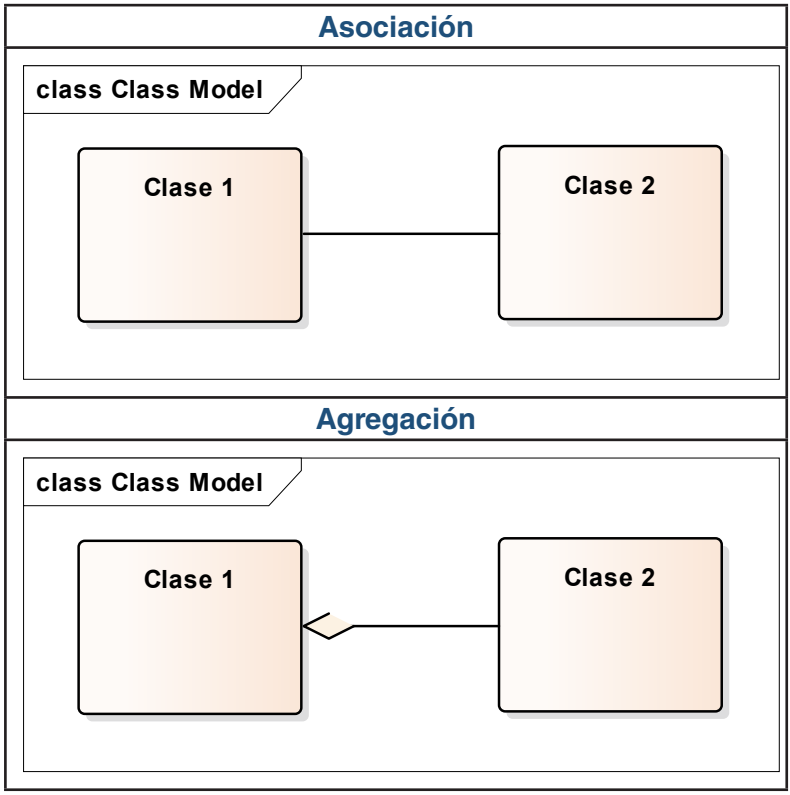


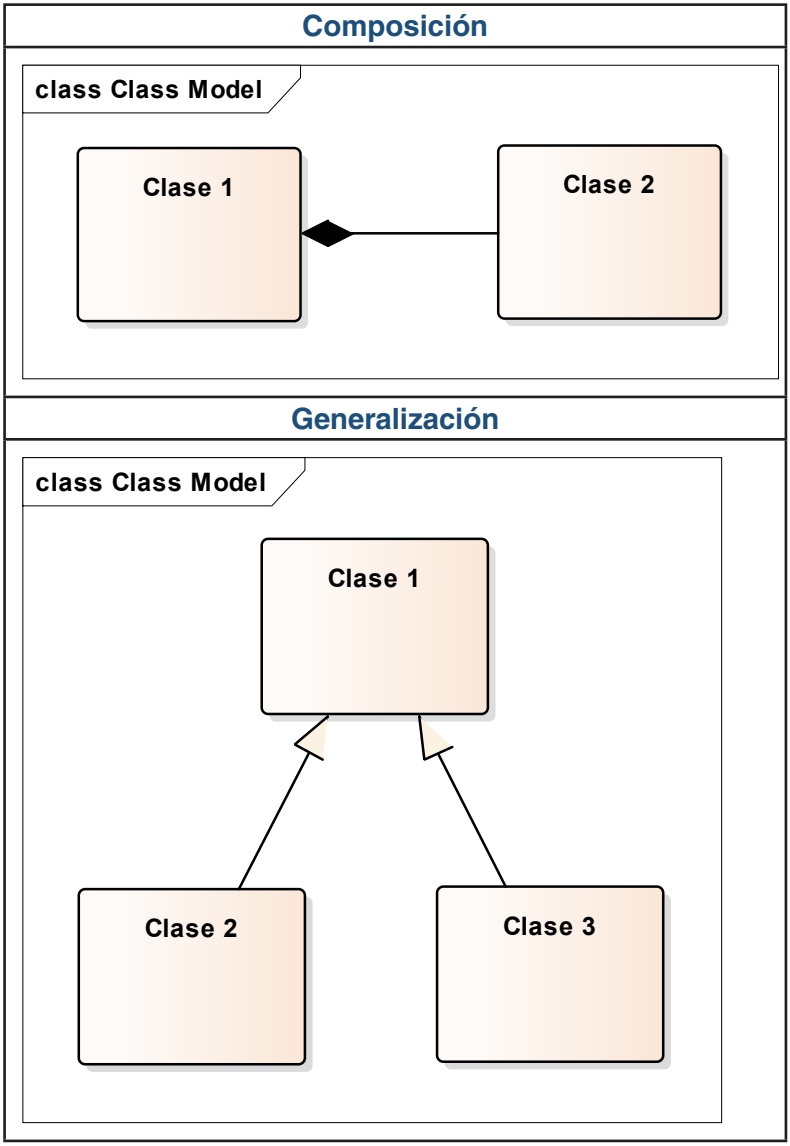
Figura 1.1. Representación UML de una clase.

Contiene notaciones para las diferentes relaciones entre clases como la asociación, que se representa

a través de una línea sólida entre las clases que puede ser unidireccional (un solo extremo con flecha) o bidireccional (sin flechas). La agregación que se representa como una línea con un diamante vacío (◊) en el extremo de la clase contenedora. La composición es un caso especial de la agregación que se representa por una línea con un diamante relleno (◆) en el extremo de la clase contenedora. La generalización que se representa por una línea con una flecha vacía apuntando a la clase base.

Tabla 1.1. Representación UML de las relaciones de asociación, agregación, composición y generalización.




Composición
class Class Model
Clase 1
Clase 2
Generalización
class Class Model
Clase 1
Clase 2
Clase 3

Los estereotipos que se utilizan en el modelado para categorizar clases y elementos UML se escriben entre «guillemets» arriba del nombre de la clase. Un ejemplo de ello es para especificar que una clase es una interfaz y se utiliza el estereotipo «interface» (Figura 1.2).

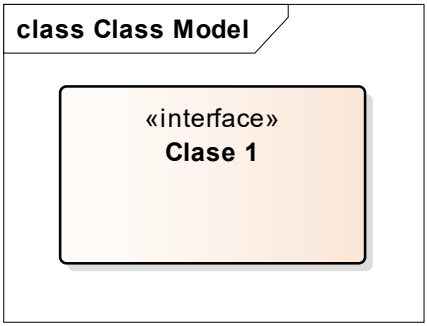


Figura 1.2. Representación UML de una clase Interfaz.

1.4. Relación de asociación, agregación y composición

En el diseño orientado a objetos, las relaciones como asociación, agregación y composición son fundamentales para representar las dependencias y estructuras entre clases. Estas relaciones permiten definir cómo se conectan y colaboran los objetos en un modelo, distinguiendo cuando la vida de los componentes depende o no del contenedor (Alemán Flores, 2025; Al-Fedaghi, 2022).

Cuando se programa en un lenguaje Orientado a Objetos como Java, la definición adecuada de las relaciones entre clases permite obtener sistemas modulares y reutilizables. Un ejemplo de estas relaciones son la asociación, agregación y composición.

La asociación es la relación entre dos clases donde una usa a la otra, pero no tienen una relación fuerte. Por ejemplo, un hospital tiene doctores que trabajan en él, pero los doctores pueden trabajar en otros hospitales también. Una de sus características es que los objetos

pueden existir de manera independiente y la relación en UML puede ser unidireccional o bidireccional.

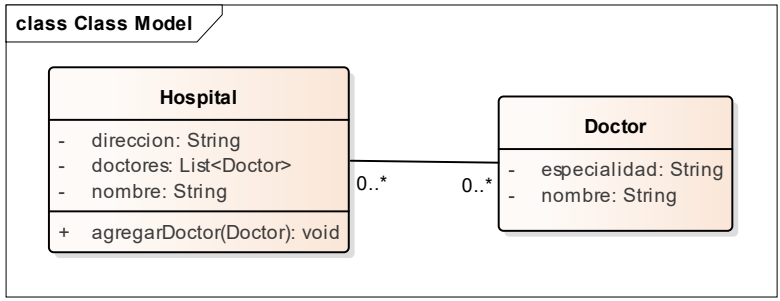


Figura 1.3. Representación UML de una relación de asociación.

Como se observa en el diagrama UML, este tipo de relación se usa cuando una clase necesita interactuar con otra, pero sin que una dependa de la otra para existir. Los objetos involucrados en la relación no se poseen mutuamente, simplemente colaboran. Se puede representar mediante una referencia a la otra clase como se muestra en el desarrollo en Java de este modelo:

```
class Doctor {
    private String nombre;
    private String especialidad;

    public Doctor(String nombre, String especialidad) {
        this.nombre = nombre;
        this.especialidad = especialidad;
    }
}
```

```
class Hospital {
    private String nombre;
    private String direccion;
    // Asociación: El hospital conoce a los doctores, pero no los posee
    private List<Doctor> doctores;

    public Hospital(String nombre, String direccion) {
        this.nombre = nombre;
        this.direccion = direccion;
        this.doctores = new ArrayList<>();
    }

    public void agregarDoctor(Doctor doctor) {
        doctores.add(doctor);
    }
}
```

En esta implementación el hospital conoce a los doctores, pero los doctores existen independientemente. Esto quiere decir que un mismo doctor podría estar asociado a múltiples hospitales. Esta relación es flexible y no hay propiedad fuerte.

La agregación es una relación "tiene un" en la que una clase contiene a otra, pero el objeto contenido puede existir independientemente del contenedor. Un ejemplo de esto es el caso de un concesionario de autos, este puede contener autos, pero los autos existen sin estar

en el concesionario. Esto no regula una dependencia de la existencia de uno con el otro por lo que resulta ser una relación débil. En UML se representa a través de un diamante vacío. Los objetos de clases pueden ser compartidos por varias instancias, significa que múltiples objetos pueden hacer referencia a un mismo objeto en memoria en lugar de crear copias separadas. Por ejemplo, en una relación de agregación, un mismo auto puede estar en diferentes concesionarios sin que su existencia dependa de ninguno en particular.

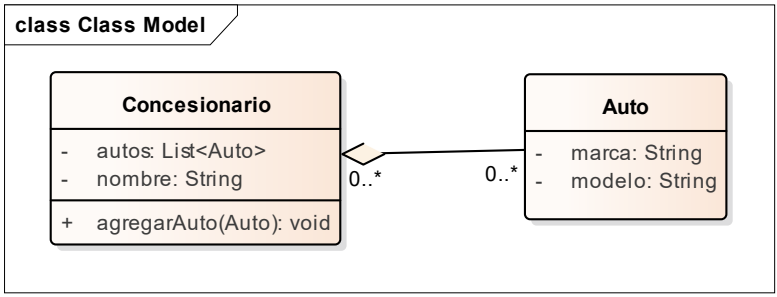


Figura 1.4. Representación UML de una relación de agregación.

Esta relación entre clases es un caso especial de asociación, donde un objeto "tiene" otro, pero este puede existir de manera independiente. Se modela usando una referencia, igual que la asociación, pero con el significado de que el objeto referenciado no es exclusivo. El desarrollo en Java de este modelo es:

```
class Auto {
    private String modelo;
    private String marca;
    public Auto(String modelo, String marca) {
        this.modelo = modelo;
        this.marca = marca;
    }
```

```
}

class Concesionario {
    private String nombre;
        // Agregación: El concesionario tiene autos, pero los autos pueden existir sin él
    private List<Auto> autos;

    public Concesionario(String nombre) {
        this.nombre = nombre;
        this.autos = new ArrayList<>();
    }

    public void agregarAuto(Auto auto) {
        autos.add(auto);
    }
}
```

En la implementación se puede observar que la clase Concesionario tiene una lista de Auto, pero los autos pueden existir sin el concesionario. Un mismo Auto podría estar en múltiples Concesionarios, lo que no ocurre en composición. Esto define débil dependencia entre el concesionario y los autos.

La composición es un tipo especial de agregación donde los objetos dependen completamente del contenedor. Si el contenedor se destruye, los objetos internos también se destruyen. Por ejemplo, una frutería

tiene frutas, pero si la frutería cierra, sus frutas dejan de estar disponibles. Se considera una relación fuerte de propiedad. Se representa con un diamante negro en UML. El ciclo de vida de los objetos está ligado al contenedor (Figura 1.5).

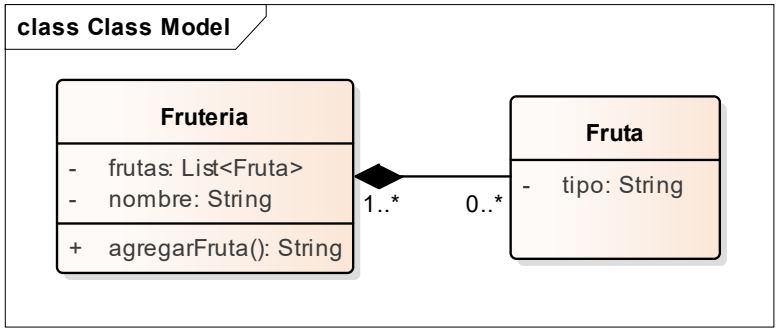


Figura 1.5. Representación UML de una relación de composición.

El desarrollo en Java de este modelo es:

```
class Fruta {
    private String tipo;

    public Fruta(String tipo) {
        this.tipo = tipo;
    }
}

class Fruteria {
    private String nombre;
    private List<Fruta> frutas;
```

```
        public Fruteria(String nombre) {
            this.nombre = nombre;
            this.frutas = new ArrayList<>();
        }

        public void agregarFruta(String tipo) {
            Fruta nuevaFruta = new Fruta(tipo);
            frutas.add(nuevaFruta);
        }
    }
```

En la Tabla 1.2 se puede observar la descripción comparativa sobre este tipo de relaciones teniendo en cuenta lo que se ha analizado hasta el momento:

Tabla 1.2. Tabla comparativa de las relaciones de asociación, agregación y composición.

Criterio	Asociación	Agregación	Composición
Definición	Relación general entre clases, donde una usa a otra, sin poseerla necesariamente.	La clase padre (principal o base) incluye instancias de otra clase. Estas instancias de clases existen por sí mismos, no importa si la clase que lo contiene es destruida.	La clase padre (principal o base) incluye instancias de otra clase. Estas instancias de clases dependen totalmente de la existencia de su clase padre y no pueden existir sin ella.

Grado de dependencia	Relación entre clases con un nivel de dependencia débil. Los objetos interactúan entre sí, pero son independientes en su ciclo de vida.	Representa un nivel medio de dependencia porque, aunque una clase contiene a otra, los objetos contenidos pueden existir de forma independiente.	Representa el nivel más alto de dependencia porque los objetos contenidos no pueden existir sin su contenedor y son parte fundamental de su estructura.
Representación en UML	Línea simple entre clases, con posible multiplicidad.	Línea con diamante vacío (◊) en el lado del contenedor.	Línea con diamante relleno (◆) en el lado del contenedor.
Ciclo de vida	Independiente entre clases.	Independiente entre clases, aunque existe una relación de propiedad.	El objeto contenido es destruido si el contenedor se destruye.
Ejemplo en la vida real	Un estudiante está inscrito en varios cursos.	Una biblioteca tiene libros, pero los libros pueden existir sin la biblioteca.	Un cuerpo humano tiene órganos, que no pueden existir sin el cuerpo.
Implementación en Java	Se usa una referencia a otra clase dentro de la clase principal.	Se usa una lista de objetos o una referencia, sin restricción de propiedad.	Se instancia directamente dentro de la clase, asegurando que la existencia del objeto dependa del contenedor.

En la tabla anterior quedan claras las definiciones y su representación sobre ejemplos concretos. Pero no siempre es claro definirlas o suele haber casos dudosos que requiere de un análisis más profundo, por eso es requerido tener en cuenta para esto el contexto que se está analizando, debido a que dos objetos pudieran variar sus relaciones entre sí en dependencia del caso de análisis.

Veamos como varía la definición del modelo de clases UML y la programación teniendo en cuenta la dependencia entre las clases Departamento y Profesor. Para ello se analizarán tres casos donde se altera la solución teniendo en cuenta un contexto diferente.

Caso 1

Suponga que un profesor de matemáticas puede participar en diferentes materias asociadas a diferentes departamentos de una universidad cualquiera. Este profesor puede enseñar álgebra en el Departamento de Ciencias Exactas para futuros matemáticos, y al mismo tiempo enseñar cálculo en el Departamento de Arquitectura para estudiantes que diseñarán puentes y edificios. Si el profesor decide dejar de dar clases en el Departamento de Ciencias Exactas, puede seguir enseñando normalmente en Arquitectura, pues su trabajo no está atado exclusivamente a un solo departamento sino a la Universidad en sí. La relación entre ambos es flexible, porque, aunque el profesor y los departamentos están conectados, su relación es débil lo que significa que son independientes entre sí. Entonces, el profesor más bien mantiene un contrato directo con la Universidad, que le permite prestar servicios en el área que lo necesite sin que deje de ser profesor cuando se mueve por estas áreas. Esto

sugiere una relación de asociación, porque la relación es opcional y no hay una dependencia fuerte entre el profesor y los departamentos. A continuación, se muestra el diagrama de clases UML que representaría esta relación entre ambos (Figura 1.6).

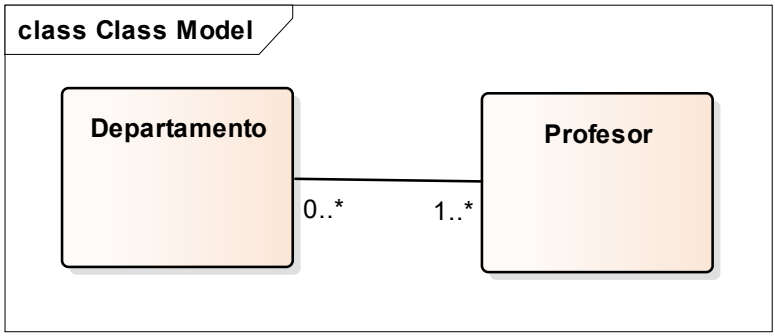


Figura 1.6. Representación UML de la relación de asociación entre Departamento y Profesor.

Caso 2

Cada departamento tiene una cantidad definida de profesores afiliados, pero estos pueden pertenecer a varios departamentos o incluso ser trasladados a otra universidad sin que el departamento desaparezca. Si un profesor se jubila, el departamento sigue existiendo sin problemas, y simplemente contrata otro docente en su lugar. En este contexto, definir una relación de agregación es lo más conveniente, porque el departamento contiene a los profesores, pero no los controla completamente, ya que estos pueden existir de forma independiente y pueden estar vinculados a otros departamentos. A continuación, se muestra el diagrama de clases UML que representaría esta relación entre ambos (Figura 1.7).

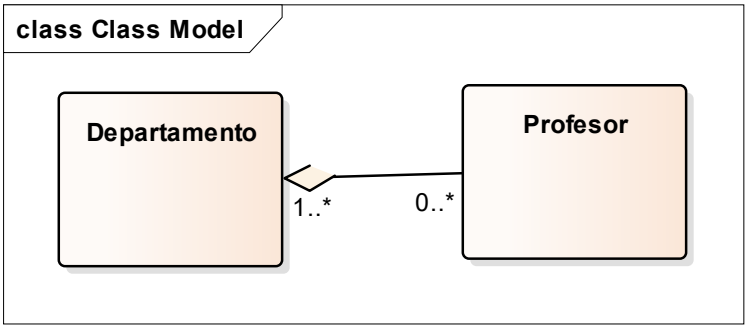


Figura 1.7. Representación UML de la relación de agregación entre Departamento y Profesor.

Caso 3

Considere que un departamento de Química tiene profesores permanentes que son parte fundamental de su estructura por el conocimiento especifico del área. Estos profesores están tan íntimamente ligados al departamento que su posición como profesores permanentes solo existe mientras el departamento exista. Si por alguna razón el Departamento de Química se cerrara, estos profesores no podrían moverse a otro departamento como Biología o Física, porque su especialización no es requerida en este contexto, además su contrato y posición están exclusivamente vinculados al Departamento de Química lo que hace que entre ellos exista una dependencia alta. Esta relación es tan fuerte que, si el departamento desaparece, los profesores pierden automáticamente su estatus de permanentes, y si quisieran trabajar en otro departamento, tendrían que pasar por todo el proceso de contratación nuevamente. Es como si el profesor permanente y el departamento fueran una sola unidad inseparable, donde uno no puede existir sin el otro. Este caso apunta a que es necesario implementar una relación de composición. A continuación, se muestra el diagrama de clases UML que representaría esta relación entre ambos.

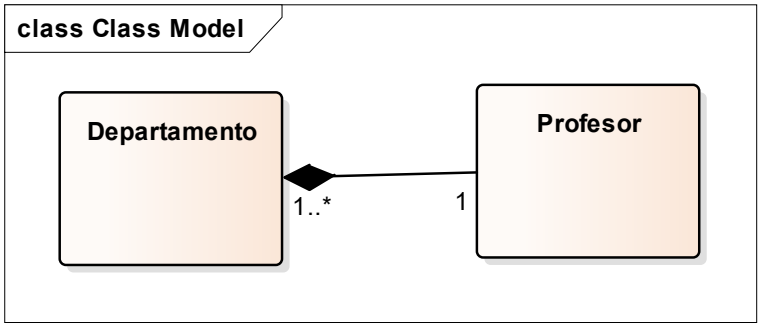


Figura 1.8. Representación UML de la relación de composición entre Departamento y Profesor.

Saber elegir la relación adecuada durante el proceso de desarrollo de software es de vital importancia para definir la adecuada arquitectura del sistema a implementar. Tener en cuenta para esto la interacción y dependencias entre sí, facilita la división correcta de objetos impactando positivamente sobre la modularidad. También el tipo de relación influye en cómo se crean y destruyen los objetos en memoria, evitando referencias innecesarias. Además, reduce el acoplamiento y aumenta la cohesión, facilitando la evolución del sistema. Al mismo tiempo, el código se vuelve más claro lo que permite evadir errores lógicos como que un objeto dependa innecesariamente de otro y evita referencias a objetos inexistentes (null pointer exceptions). Esto hace que se escriba código más limpio, modular y eficiente, garantizando una correcta gestión del ciclo de vida de los objetos.

Caso de estudio Gestión de Hospital

A continuación, se define un caso de estudio que permite mostrar una solución que integra estas tres relaciones. El caso de estudio se enfoca a la gestión que se realiza en un Hospital y sus Áreas.

Un hospital está compuesto por varias áreas médicas (como pediatría, cardiología, emergencias y otras).

Cada área médica tiene doctores asignados, pero estos pueden trabajar en varias áreas. Cada área médica pertenece a un hospital, pero puede existir sin él. Dentro de cada área médica, hay equipos médicos (máquinas especializadas) que pertenecen exclusivamente a esa área y si el área desaparece, los equipos también lo hacen.

Relaciones que pueden sacarse de la descripción anterior:

- Relación de asociación: Se define cuando un doctor puede trabajar en varias áreas médicas y viceversa.
- Relación de agregación: Un hospital tiene varias áreas médicas, pero un área podría pertenecer a otro hospital.
- Relación de composición: Un área médica contiene los equipos médicos requeridos, los cuales no existen sin el área.

El modelo de clases UML (Figura 1.9) que se propone es:

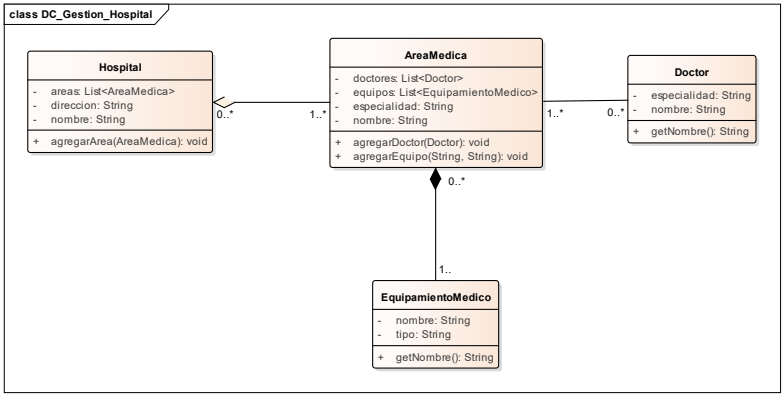


Figura 1.9. Representación UML del caso de estudio 1: Gestión de Hospital.

El código en java para el UML sugerido es:

Lo primero es entender que según el modelo UML, los doctores existen por sí mismos y pueden estar en múltiples áreas médicas.

```
class Doctor {
    private String nombre;
    private String especialidad;

    public Doctor(String nombre, String especialidad) {
        this.nombre = nombre;
        this.especialidad = especialidad;
    }

    public String getNombre() {
        return nombre;
    }
}
```

El EquipamientoMedico no puede existir sin AreaMedica, por lo que su instancia se gestiona dentro de la composición.

```
class EquipamientoMedico {
    private String nombre;
    private String tipo;

    public EquipamientoMedico(String nombre, String tipo) {
```

```
        this.nombre = nombre;
        this.tipo = tipo;
    }

    public String getNombre() {
        return nombre;
    }
}
```

El AreaMedica tiene doctores (asociación, porque un doctor puede trabajar en varias áreas) y también posee equipos médicos (composición, porque los equipos solo existen dentro del área).

```
class AreaMedica {
    private String nombre;
    private String especialidad;
     private List<Doctor> doctores; // Asociación (Múltiples doctores pueden estar en el área)
     private List<EquipamientoMedico> equipos; // Composición (Los equipos pertenecen al área)

    public AreaMedica(String nombre, String especialidad) {
        this.nombre = nombre;
        this.especialidad = especialidad;
        this.doctores = new ArrayList<>();
        this.equipos = new ArrayList<>();
    }
```

```
    public void agregarDoctor(Doctor doctor) {
        doctores.add(doctor);
    }

    public void agregarEquipo(String nombre, String tipo) {
        equipos.add(new EquipamientoMedico(nombre, tipo));
    }
}
```

Hospital tiene áreas médicas, pero estas podrían existir fuera de él (agregación).

```
class Hospital {
    private String nombre;
    private String direccion;
   // Agregación (Un hospital tiene áreas, pero estas pueden existir sin él)
    private List<AreaMedica> areas;

    public Hospital(String nombre, String direccion) {
        this.nombre = nombre;
        this.direccion = direccion;
        this.areas = new ArrayList<>();
    }
```

```
    public void agregarArea(AreaMedica area) {
        areas.add(area);
    }
}
```

Ejemplo de uso implementado en la clase principal:

```
public class Main {
    public static void main(String[] args) {
        // Crear un hospital
        Hospital hospital = new Hospital("Hospital Central", "Av. Principal 123");

        // Crear áreas médicas
        AreaMedica pediatria = new AreaMedica("Pediatría", "Cuidado infantil");
        AreaMedica cardiologia = new AreaMedica("Cardiología", "Cuidado del corazón");

        // Agregar áreas al hospital (Agregación)
        hospital.agregarArea(pediatria);
        hospital.agregarArea(cardiologia);

        // Crear doctores
        Doctor doctor1 = new Doctor("Dr. Pérez", "Pediatría");
        Doctor doctor2 = new Doctor("Dra. Gómez", "Cardiología");
```

```
        // Asociar doctores a áreas médicas (Asociación)
        pediatria.agregarDoctor(doctor1);
        cardiologia.agregarDoctor(doctor2);

        // Agregar equipamiento a áreas médicas (Composición)
        pediatria.agregarEquipo("Incubadora", "Neonatal");
        cardiologia.agregarEquipo("Electrocardiógrafo", "Diagnóstico");

        // Mostrar mensaje
        System.out.println("Sistema del hospital configurado correctamente". );
    }
}
```

En el modelo propuesto se han aplicado principios de alta cohesión y bajo acoplamiento, fundamentales en el diseño de sistemas orientados a objetos. Veamos cómo se logran estos principios en cada parte del modelo:

La alta cohesión implica que cada clase tiene una responsabilidad clara y bien definida, evitando que una sola clase haga más de lo que le toca. Por ejemplo:

- Un doctor maneja información y acciones relacionadas con un médico.
- El equipamiento médico solo maneja información sobre los equipos.

- El Área Médica se encarga de gestionar a los doctores y a los equipos, pero no maneja otras responsabilidades del hospital.
- El hospital solo administra las áreas médicas.

En las clases de define el ocultamiento de datos a través de los atributos privados, permitiendo modificar datos solo a través de métodos controlados por la clase. Para asegurar que las áreas médicas puedan administrar elementos que le competen utiliza los métodos agregarDoctor() y agregarEquipo(). Además, aplica el principio de diseño que consiste en dividir un sistema en partes independientes, donde cada una maneja una única responsabilidad o aspecto del software como se muestra en la clase Doctor que no sabe nada sobre la clase Hospital, solo sobre la clase AreaMedica evadiendo las dependencias innecesarias. Además, la clase EquipamientoMedico no necesita saber sobre Hospital o Doctor, solo sobre las AreaMedica. El Hospital no maneja doctores ni equipos directamente, solo áreas médicas. Este tipo de modelo permite que el código esté dividido en módulos, que sea fácil de entender, mantener y extender.

Por otro lado, el bajo acoplamiento significa que las clases tienen la menor dependencia posible entre sí, de modo que los cambios en una afecten lo menos posible a otras. Un ejemplo de ello es el uso de Listas en lugar de referencias directas como en la clase Hospital que tiene la información de las áreas médicas a través de una lista de ellas (List<AreaMedica>), pero no gestiona su ciclo de vida. La clase AreaMedica, contiene una lista de doctores a través de List<Doctor>, pero un Doctor no almacena referencias a las áreas médica. La clase EquipamientoMedico solo existe dentro de la clase AreaMedica a través de una relación de composición, siendo AreaMedica la que lo administra internamente.

Además, se implementan métodos específicos para interactuar entre clases, un ejemplo de esto es la clase Hospital que no manipula directamente las áreas médicas, en vez de ello contiene un método que se encarga de ello como agregarArea(). También, en la clase AreaMedica no accede directamente a un Doctor, sino que a través del método agregarDoctor() se logra la gestión del mismo. Igualmente, el AreaMedica solo crea y administra EquipamientoMedico, asegurando que no haya referencias externas a ellos. Asimismo, minimiza las dependencias y las controla, esto se visualiza en el problema cuando el Doctor puede existir sin Hospital y sin el área médica, lo que permite ser reutilizable para otros sistemas o módulos.

También, el EquipamientoMedico está completamente dentro de AreaMedica, evitando dependencias con otras clases. Hospital no tiene referencias directas a Doctor ni a EquipamientoMedico, solo a AreaMedica. Esto trae como beneficio que si se requiere modificar la forma en que se manejan los doctores o equipos, solo se afectan las clases directamente relacionadas.

1.5. Relación de generalización y especialización

Las relaciones de generalización y especialización permiten expresar jerarquías de clases donde una clase más general (superclase) define atributos y comportamientos comunes que son heredados por clases más específicas (subclases), facilitando la reutilización, modularidad y cohesión del sistema (Shah y Grant, 2022). La generalización se representa en UML como una relación is-a que indica que una subclase “es un tipo de” su superclase, mientras que la especialización define cómo se concretan y extienden los comportamientos en contextos particulares del dominio (Shah y Grant, 2024). Esta capacidad de estructurar modelos jerárquicos con claridad y

formalidad resulta esencial no solo para la comprensión conceptual, sino también para la evolución controlada de sistemas complejos, siendo un tema de estudio activo en investigaciones de ingeniería de software y verificación de modelos UML.

La generalización es un concepto de la POO que agrupa propiedades comunes de múltiples clases en una única clase más general, llamada superclase (clase padre, clase base). Es un principio que se aplica al diseño de las clases, que busca no repetir el código al identificar características comunes entre varias clases y agruparlas en una superclase que las contenga. Esta superclase actúa como una plantilla donde se definen los atributos y los comportamientos básicos que serán utilizados por todas las subclases relacionadas. Por su parte, la especialización es lo contrario, permite crear clases más específicas que, además de heredar las características de la superclase, añaden sus propias características y métodos únicos. Lo que permite crear una jerarquía clara donde las características generales fluyen desde la clase padre hacia las clases hijas, mientras que los detalles específicos se agregan en las subclases. Esto permite incrementar la mantenibilidad del código y reducir su duplicación. Además, se logra representar más acorde a las relaciones naturales entre diferentes tipos de objetos reflejando cómo en el mundo real las cosas pueden compartir características comunes mientras mantienen sus propias particularidades.

Para visualizar este tipo de relaciones, imagine que se está desarrollando un software para un estudio fotográfico profesional que maneja diferentes tipos de sesiones fotográficas como para bodas, graduación y la corporativa, las cuales se consideran clases para el modelo que se quiere implementar. Aunque cada tipo de sesión tiene sus particularidades, todas comparten elementos básicos como:

- La duración de la sesión.
- El precio base.
- La cantidad de fotografías entregables.
- La fecha programada.
- El fotógrafo asignado.

También comparten operaciones como:

- Calcular el precio final.
- Programar de sesiones
- Puede actualizar el fotógrafo asignado.

Es inadecuado analizar las clases como si no estuvieran relacionadas, definiendo cada atributo y operaciones en cada una de ellas por separado, sin tener en cuenta su tronco común. Aunque podría funcionar, haría que el código fuese repetitivo, no reutilizable, afectando la modularidad y al mismo tiempo aumentaría significativamente la cantidad de código y también el riesgo de cometer errores al mantener múltiples copias del mismo código. Para evitar esto se debe aplicar los conceptos de generalización y especialización creando una superclase llamada "SesiónFotográfica" que contenga todos estos elementos comunes y que no sean repetidos en cada subclase, la cual constituiría una generalización. Las clases hijas son las que heredarán estas características y añadirán sus propias operaciones aplicando la definición de especialización como se muestra a continuación:

- Para la clase SesiónBoda deben ser añadido sus atributos específicos como la cantidad de invitados, si incluye album físico o no y la ubicación de la ceremonia, además de sus

propias operaciones como la de coordinar con el planificador y agregar sesión pre boda.

- Para la clase SesiónGraduación debe incluir los atributos como facultad y la cantidad de graduados. Además, incluir los métodos específicos como organizar los grupos y gestionar las togas.
- Para la clase SesiónCorporativa debe incluir los atributos como empresa del cliente y el tipo de evento. También las operaciones propias como aplicar la marca de la empresa y generar el portafolio corporativo.

El diagrama de clases UML que responde a una generalización-especialización (Figura 1.10) del problema es el siguiente:

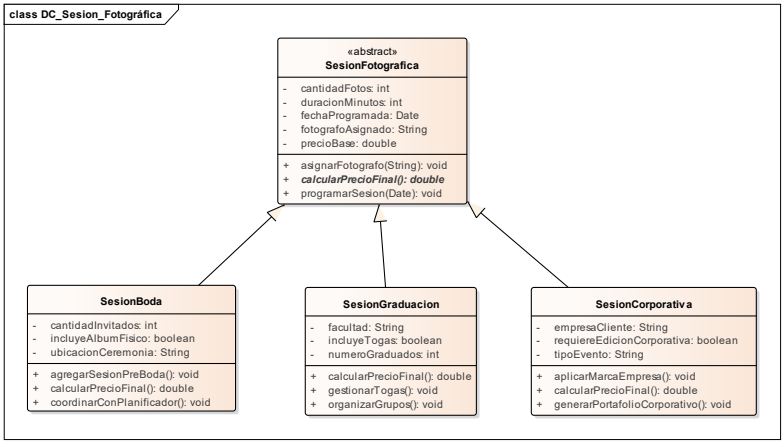


Figura 1.10. Representación UML de la relación generalización-especialización del ejemplo Sesión Fotográfica.

Este diagrama muestra la jerarquía de clases, lo que facilita la comprensión del sistema antes de comenzar a codificarlo en cualquier lenguaje de programación. Para los desarrolladores constituye un identificador

rápido de las relaciones de herencia entre las subclases como especializaciones y la clase base como su generalización, lo que ayuda a evitar redundancias. Además, el diagrama UML favorece a la documentación visual del sistema, facilitando la comunicación entre los miembros del equipo de desarrollo y stakeholders. El código suele estructurarse de forma más organizado y consistente, reduciendo la posibilidad de errores y mejorando su calidad. Por último, actúa como un contrato de diseño que asegura que la implementación seguirá los principios de la POO, específicamente la herencia y el encapsulamiento.

En el caso de no aplicar estos conceptos, cualquier modificación como cambiar el cálculo del precio base o la forma de programar sesiones, requeriría actualizar el código en las tres clases hijas, lo que aumentaría la probabilidad de inconsistencias y duplicidad de código. Además, sin la estructura jerárquica que proporciona la generalización, no se podría aplicar el concepto de polimorfismo al no poder crear una lista o arreglo de tipo SesionFotografica que pudiera contener cualquier tipo de sesión, lo que complicaría la implementación de funcionalidades como la gestión de calendarios o la generación de reportes. También dificultaría la extensibilidad del sistema, ya que añadir un nuevo tipo de sesión fotográfica requeriría copiar y adaptar toda la lógica común nuevamente, en lugar de simplemente heredar de la clase base y añadir las especializaciones necesarias. Por último, el código sería poco intuitivo para otros desarrolladores, ya que no reflejaría la relación natural y lógica que existe entre los diferentes tipos de sesiones fotográficas, perdiendo así una importante capa de documentación implícita que proporciona el diseño orientado a objetos.

El código en java que responde al diagrama UML se expresa a continuación:

```
// Clase base
public abstract class SesionFotografica {
    private int duracionMinutos;
    private double precioBase;
    private int cantidadFotos;
    private Date fechaProgramada;
    private String fotografoAsignado;

    // Constructor
    public SesionFotografica(int duracionMinutos, double
precioBase, int cantidadFotos) {
        this.duracionMinutos = duracionMinutos;
        this.precioBase = precioBase;
        this.cantidadFotos = cantidadFotos;
    }

    // Métodos getters y setters
    public int getDuracionMinutos() { return duracionMinutos;
}
    public void setDuracionMinutos(int duracionMinutos) {
this.duracionMinutos = duracionMinutos; }
    public double getPrecioBase() { return precioBase; }
    public void setPrecioBase(float precioBase) { this.pre-
cioBase = precioBase; }
    public int getCantidadFotos() { return cantidadFotos; }
```

```
    public void setCantidadFotos(int cantidadFotos) { this.
cantidadFotos = cantidadFotos; }


    // Métodos comunes
    public abstract double calcularPrecioFinal();


    public void programarSesion(Date fecha) {
        this.fechaProgramada = fecha;
      System.out.println("Sesión programada para: " + fecha);
    }


    public void asignarFotografo(String fotografo) {
        this.fotografoAsignado = fotografo;
        System.out.println("Fotógrafo asignado: " + fotografo);
    }
}
```

En la clase anterior "SesionFotografica", se utiliza la palabra clave abstract en el método calcularPrecioFinal() para indicar que no pueden ser instanciado directamente, sino que debe ser implementado o heredado por otras clases. Por eso el método calcularPrecioFinal() no tiene una definición en la superclase sino que se sobrescribirá en las clases hijas con definiciones diferentes.

```
// Clase SesionBoda
public class SesionBoda extends SesionFotografica {
    private int cantidadInvitados;
```

```
    private boolean incluyeAlbumFisico;
    private String ubicacionCeremonia;

    public SesionBoda(int duracionMinutos, double precioBase, int cantidadFotos,
                      int cantidadInvitados, boolean incluyeAlbumFisico) {
        super(duracionMinutos, precioBase, cantidadFotos);
        this.cantidadInvitados = cantidadInvitados;
        this.incluyeAlbumFisico = incluyeAlbumFisico;
    }

    @Override
    public double calcularPrecioFinal() {
        double precioFinal = getPrecioBase();
        if (incluyeAlbumFisico) precioFinal += 200;
        if (cantidadInvitados > 100) precioFinal += 300;
        return precioFinal;
    }

    public void coordinarConPlanificador() {
        System.out.println("Coordinando detalles con el planificador de bodas");
    }
```

```
    public void agregarSesionPreBoda() {
        System.out.println("Sesión pre-boda agregada al paquete");
    }
}
```

En la clase anterior "SesionBoda", se utiliza la palabra clave extends para establecer la herencia, de la misma forma que verán en las posteriores clases del modelo que heredan con la clase "SesionFotografica". También se verifica el uso de super() en los constructores de las clases hijas para llamar al constructor de la clase padre. Y se utiliza @Override para indicar que un método está sobrescribiendo al de la clase padre.

```
// Clase SesionGraduacion
public class SesionGraduacion extends SesionFotografica {
    private String facultad;
    private int numeroGraduados;
    private boolean incluyeTogas;

    public SesionGraduacion(int duracionMinutos, double precioBase, int cantidadFotos,
                            String facultad, boolean incluyeTogas) {
        super(duracionMinutos, precioBase, cantidadFotos);
        this.facultad = facultad;
        this.incluyeTogas = incluyeTogas;
    }
```

```
    @Override
    public double calcularPrecioFinal() {
        float precioFinal = getPrecioBase();
        if (incluyeTogas) precioFinal += 50;
        return precioFinal;
    }

    public void organizarGrupos() {
        System.out.println("Organizando grupos para fotos de
graduación");
    }

    public void gestionarTogas() {
        if (incluyeTogas) {
            System.out.println("Gestionando togas para la se-
sión");
        }
    }
}

// Clase SesionCorporativa
public class SesionCorporativa extends SesionFotografica {
    private String empresaCliente;
    private String tipoEvento;
```

```
    private boolean requiereEdicionCorporativa;

    public SesionCorporativa(int duracionMinutos, double
precioBase, int cantidadFotos,
                    String empresaCliente, boolean requiere-
EdicionCorporativa) {
        super(duracionMinutos, precioBase, cantidadFotos);
        this.empresaCliente = empresaCliente;
        this.requiereEdicionCorporativa = requiereEdicionCor-
porativa;
    }

    @Override
    public float calcularPrecioFinal() {
        float precioFinal = getPrecioBase();
        if (requiereEdicionCorporativa) precioFinal += 150;
        return precioFinal;
    }

    public void aplicarMarcaEmpresa() {
        System.out.println("Aplicando marca de la empresa en
las fotos");
    }

    public void generarPortafolioCorporativo() {
```

```
        System.out.println("Generando portafolio corporativo");
    }
}
```

Un ejemplo de uso en el método principal:

```
public class Main {
    public static void main(String[] args) {
        // Crear una sesión de boda
        SesionBoda boda = new SesionBoda(240, 1000.0, 200, 150, true);
        boda.asignarFotografo("Juan Pérez");
        boda.programarSesion(new Date());
        System.out.println("Precio final boda: $" + boda.calcularPrecioFinal());

        // Crear una sesión de graduación
        SesionGraduacion graduacion = new SesionGraduacion(120, 500.0, 100, "Ingeniería", true);
        graduacion.organizarGrupos();
        System.out.println("Precio final graduación: $" + graduacion.calcularPrecioFinal());

        // Crear una sesión corporativa
        SesionCorporativa corporativa = new SesionCorporativa(180, 800.0, 150, "TechCorp", true);
        corporativa.aplicarMarcaEmpresa();
        System.out.println("Precio final corporativa: $" + corpo-
```

```
rativa.calcularPrecioFinal());
    }
}
```

El método main anterior demuestra el uso práctico de la jerarquía de clases de sesiones fotográficas. En él se crean tres instancias diferentes: una SesionBoda con duración de 240 minutos y precio base de $1000, una SesionGraduacion de 120 minutos con precio base de $500, y una SesionCorporativa de 180 minutos con precio base de $800, donde cada tipo de sesión muestra sus características específicas y cálculos de precio personalizados. De la misma forma, muestra cómo cada tipo de sesión puede utilizar tanto los métodos heredados de la clase base (asignarFotografo y programarSesion) como sus métodos específicos (coordinarConPlanificador para bodas u organizarGrupos para graduaciones), demostrando así el polimorfismo (concepto que se explicará posteriormente en detalle) y la herencia en acción. Este método principal actúa como una demostración práctica permitiendo visualizar cómo los diferentes tipos de sesiones fotográficas pueden ser creados y manipulados, mostrando los precios finales calculados según las características específicas de cada tipo de sesión.

1.6. Clases abstractas

Las clases abstractas constituyen un mecanismo fundamental dentro de la programación orientada a objetos, ya que permiten definir modelos conceptuales generales que establecen contratos de comportamiento sin imponer una implementación concreta. A través de la abstracción, es posible separar el “qué hace” un objeto del “cómo lo hace”, favoreciendo diseños más flexibles, extensibles y alineados con el dominio del problema. En

lenguajes como Java, las clases abstractas se utilizan para capturar características comunes entre distintos tipos de objetos y forzar a las subclases a implementar comportamientos específicos, lo que reduce la duplicación de código y mejora la mantenibilidad del sistema. Diversos estudios recientes destacan que el uso adecuado de abstracción y clases abstractas no solo simplifica el diseño estructural, sino que también fortalece la calidad del software y la comprensión del modelo por parte de los desarrolladores, especialmente en contextos educativos y sistemas de mediana y gran escala (Lavand y Pawar, 2025; Vincent Gumonan et al., 2024).

Alineado a esta descripción, este tipo de clases definen un modelo a utilizar para otras clases, definiendo métodos abstractos no implementables y métodos concretos que si se implementan y pueden ser reutilizados por las clases hijas. Los métodos abstractos deben ser construido por las subclases que se especializan de la clase abstracta. Actúa como una plantilla que propone un diseño común, pero no puede ser instanciada directamente. Se declara la clase en Java agregando la palabra clave abstract. Por otro lado, los métodos abstractos en la clase abstracta definen un contrato que obliga a las subclases a desarrollar una implementación específica por cada una de ellas. Se declara con la palabra clave abstract y su definición termina con un punto y coma; en lugar de un bloque de código. Finalmente, en los diagramas UML, los métodos abstractos se escriben en cursiva dentro de la clase abstracta.

Retomando el ejemplo anterior relacionado con las sesiones fotográficas, se definió la clase abstracta con el nombre SesionFotografica, debido a que conceptualmente, una “sesión fotográfica” en sí misma es un concepto abstracto, no existe una sesión

fotográfica genérica, siempre será de un tipo específico (boda, graduación, corporativa y otros según el caso en la vida real). Esto se refleja perfectamente en una clase que no puede ser instanciada directamente. Además, el método definido como abstracto calcularPrecioFinal(), necesita comportamientos específicos para cada tipo de sesión. Por ejemplo, una boda tiene consideraciones de precio diferentes a una sesión corporativa. La clase abstracta nos permite definir que este cálculo debe existir (declarándolo método abstracto) pero deja la implementación específica a cada subclase. En el caso de no tener en cuenta esta definición en el ejemplo, se puede crear instancias de SesionFotografica directamente, lo cual no tendría sentido en nuestro dominio del problema y podría llevar a errores o inconsistencias en el sistema. Por ejemplo, errores de lógica de negocio al crear una SesionFotografica genérica, ¿cómo calcularía el precio final?, no habría una implementación clara del método calcularPrecioFinal() y por tanto se podrían crear sesiones "incompletas" que no siguen ningún modelo de negocio válido.

También se violaría el principio de Sustitución de Liskov, esto quiere decir que una sesión fotográfica genérica no podría sustituir adecuadamente a sus subtipos incumpliendo con los contratos específicos de cada tipo de sesión. También impactaría en la evolución del sistema pues agregar nuevos tipos de sesiones requeriría modificar más código. Además, los datos tendrían dificultades, si instanciamos como SesionFotografica sesionGenerica = new SesionFotografica(/*...*/);

¿Qué pasaría con los datos específicos como cantidad de invitados para una boda, número de graduados, marca corporativa? Estos datos no tendrían lugar en una sesión genérica.

Por eso es de vital importancia antes de codificar, tener una idea clara de los conceptos que se están tratando. Una vez más, el modelado inicial se impone para acotar los requerimientos adecuados del sistema. Con el ejemplo discutido, se puede notar que su correcto análisis fuerza a que toda sesión fotográfica sea de un tipo específico, garantiza que cada tipo implemente su lógica necesaria, mantiene la integridad del modelo de negocio, facilita el mantenimiento y la evolución del sistema, reduce la posibilidad de errores de programación y mejora la claridad del código y su intención.

A continuación, se analiza un caso de estudio relacionado con la gestión hotelera, enfocándose en la oferta de tipos de habitaciones que contienen. Donde se aplicará el concepto de clases y métodos abstractos y se desarrollará el código en Java que responde a este modelo.

Caso de estudio sobre Gestión Hotelera: Un hotel en la playa requiere gestionar sus diferentes tipos de habitaciones que varían según su ubicación, comodidades y servicios. El hotel maneja tres categorías principales de habitaciones:

- Habitación estándar, que pueden incluir o no balcón.
- Habitación con vistas al mar, que pueden estar ubicadas en diferentes pisos y algunas cuentan con acceso directo a la playa.
- Habitación suite, que pueden estar equipadas con jacuzzi y cocineta.

Cada tipo de habitación debe calcular su precio final de manera diferente según sus características específicas:

- Las habitaciones estándar aumentan su precio base en un 20% si tienen balcón.
- Las habitaciones con vistas al mar incrementan su valor según el piso en que se encuentren (10% por piso) y suman un 30% adicional si tienen acceso directo a la playa.
- Las habitaciones suites aumentan un 40% si incluyen jacuzzi y un 20% si cuentan con cocineta.

Además, cada categoría de habitación requiere un proceso de preparación específico:

- Las habitaciones estándar requieren una limpieza básica y atención especial al balcón si lo tienen.
- Las habitaciones con vistas al mar requieren una limpieza premium que incluye los ventanales y el acceso a la playa cuando corresponde.
- Las habitaciones suites demandan una limpieza premium plus que incluye el mantenimiento del jacuzzi y la limpieza de la cocineta según sus características.

El hotel necesita mantener un registro del número de habitación, precio base y estado de ocupación para todas las habitaciones, independientemente de su tipo.

Análisis sobre el caso de estudio:

En el caso de estudio se puede identificar el concepto Habitación como abstracto base, ya que no existe una habitación genérica en el hotel, sino que cada una de ellas se especializa en un tipo específico, estándar, vista al mar o suite. Las habitaciones, a pesar de ser de tipos diferentes, comparten características, pero tienen comportamientos específicos.

Los atributos comunes que se definen son:

- El número de habitación que constituiría el identificador único de la habitación.
- El precio base que es el precio inicial antes de aplicar modificadores según el tipo de habitación.
- Si la habitación está ocupada o no que proporciona el estado de ocupación de la habitación.

Estos atributos son necesarios para todas las habitaciones independientemente de su tipo.

Los métodos que deben ser abstractos son:

- Para calcular el precio final donde cada tipo de habitación se calcula diferente.
  - En las habitaciones estándar se agrega al valor el 20% si tiene balcón.
  - En habitaciones con vistas al mar se agrega al valor el 10% por piso, y 30% más si tiene acceso a la playa.
  - En las habitaciones Suite se le agrega el 40% si tiene jacuzzi y 20% si agrega cocineta.
- Para habilitar la habitación se utilizan protocolos específicos para cada tipo de habitación.
  - En las habitaciones estándar la limpieza que se realiza es básica.
  - En las habitaciones con vistas al mar la limpieza que se realiza es premium.
  - En las habitaciones suite se realiza una limpieza premium plus.

Se identifica en el problema las siguientes especializaciones a tener en cuenta:

- Para habitaciones tipo estándar, ya que contiene un atributo adicional que es si tiene balcón o no. Y se deben tener algunas consideraciones específicas como el cálculo simple del precio y la preparación básica a tener en cuenta.
- Para las habitaciones con vistas al mar, que adiciona los atributos el número de piso en el que está, si tiene o no acceso directo a la playa. Además, se debe considerar un cálculo del valor más complejo teniendo en cuenta su ubicación y las vistas.
- Para habitaciones tipo suite se añaden atributos como si tiene jacuzzi o no y si tiene cocineta o no. También se debe hacer el cálculo con múltiples modificadores y la habilitación de estas sugiere una limpieza más exhaustiva.

Las especializaciones descritas sugieren que exista una clase abstracta nombrada Habitación, que sería la clase padre o super clase, y otras tres clases hijas o subclases que heredan de habitación que se nombrarían como HabitacionEstandar, HabitacionVistaMar y Suite. Este análisis garantiza una extensibilidad futura por si se requiere agregar nuevos tipos de habitaciones, modificadores de precio adicionales, protocolos de preparación extendidos y otros servicios adicionales.

Luego del análisis realizado, se propone como solución el diagrama de clases UML (Figura 1.11):

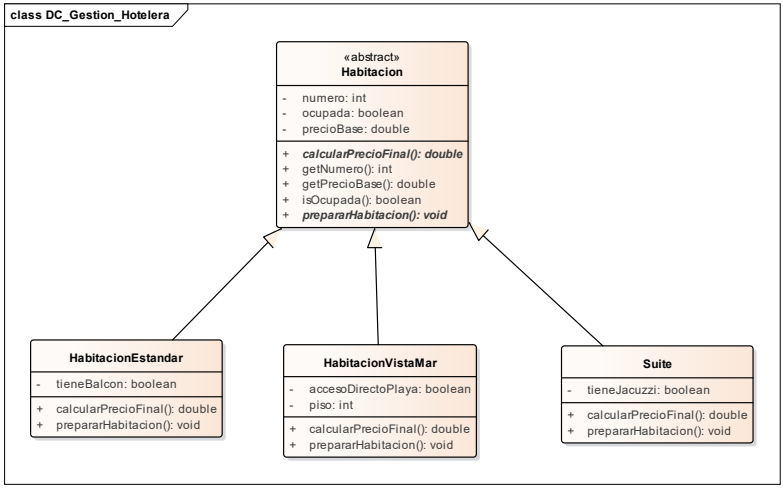


Figura 1.11. Representación UML de la relación generalización-especialización del ejemplo Gestión Hotelera.

Teniendo en cuenta el diagrama de clases, el código en java quedaría:

```
// Clase abstracta base
public abstract class Habitacion {
    private int numero;
    private double precioBase;
    private boolean ocupada;

    public Habitacion(int numero, double precioBase) {
        this.numero = numero;
        this.precioBase = precioBase;
        this.ocupada = false;
    }
```

```
    // Métodos concretos
    public int getNumero() {
        return numero;
    }

    public double getPrecioBase() {
        return precioBase;
    }

    public boolean isOcupada() {
        return ocupada;
    }

    // Métodos abstractos
    public abstract double calcularPrecioFinal();
    public abstract void prepararHabitacion();
}

// Implementación para habitación estándar
public class HabitacionEstandar extends Habitacion {
    private boolean tieneBalcon;

    public HabitacionEstandar(int numero, double precioBa-
```

```
se, boolean tieneBalcon) {
        super(numero, precioBase);
        this.tieneBalcon = tieneBalcon;
    }

    @Override
    public double calcularPrecioFinal() {
        return tieneBalcon ? getPrecioBase() * 1.2 : getPrecio-
Base();
    }

    @Override
    public void prepararHabitacion() {
         System.out.println("Preparando habitación estándar "
+ getNumero());
        System.out.println("- Limpieza básica");
        System.out.println("- Cambio de sábanas");
        if (tieneBalcon) {
            System.out.println("- Limpieza de balcón");
        }
    }
}

// Implementación para habitación con vistas al mar
public class HabitacionVistaMar extends Habitacion {
```

```
    private int piso;
    private boolean accesoDirectoPlaya;

   public HabitacionVistaMar(int numero, double precioBase,
int piso, boolean accesoDirectoPlaya) {
        super(numero, precioBase);
        this.piso = piso;
        this.accesoDirectoPlaya = accesoDirectoPlaya;
    }

    @Override
    public double calcularPrecioFinal() {
        double precio = getPrecioBase();
        precio += (piso * 10); // Mayor precio por piso más alto
        if (accesoDirectoPlaya) {
            precio *= 1.3; // 30% adicional por acceso a playa
        }
        return precio;
    }

    @Override
    public void prepararHabitacion() {
         System.out.println("Preparando habitación vista mar "
+ getNumero());
        System.out.println("- Limpieza premium");
```

```
        System.out.println("- Cambio de sábanas de alta calidad");
        System.out.println("- Limpieza de ventanales");
        if (accesoDirectoPlaya) {
            System.out.println("- Limpieza de acceso a playa");
        }
    }
}

// Implementación para suite
public class Suite extends Habitacion {
    private boolean tieneJacuzzi;
    private boolean tieneCocineta;

    public Suite(int numero, double precioBase, boolean tieneJacuzzi, boolean tieneCocineta) {
        super(numero, precioBase);
        this.tieneJacuzzi = tieneJacuzzi;
        this.tieneCocineta = tieneCocineta;
    }

    @Override
    public double calcularPrecioFinal() {
        double precio = getPrecioBase();
        if (tieneJacuzzi) precio *= 1.4; // 40% adicional por
```

```
jacuzzi
        if (tieneCocineta) precio *= 1.2; // 20% adicional por
cocineta
        return precio;
    }

    @Override
    public void prepararHabitacion() {
        System.out.println("Preparando suite " + getNumero());
        System.out.println("- Limpieza premium plus");
        System.out.println("- Cambio de sábanas de lujo");
        if (tieneJacuzzi) {
            System.out.println("- Mantenimiento de jacuzzi");
        }
        if (tieneCocineta) {
            System.out.println("Limpieza de cocineta");
        }
    }
}
```


1.7. Herencia múltiple

La herencia múltiple en la programación orientada a objetos permite a una clase derivada heredar atributos y métodos de más de una superclase. Esto hace que una misma entidad puede combinar comportamientos de distintas clases. Sin embargo, su uso también introduce desafíos, como la ambigüedad en la

resolución de métodos cuando distintas superclases poseen implementaciones similares.

En el diagrama de clases UML, la herencia múltiple se representa mediante relaciones de generalización, las cuales se indican con flechas que tienen un triángulo vacío apuntando hacia las clases bases. Debido a los conflictos de herencia y la dificultad en la gestión de dependencias, algunos lenguajes como Java evitan su implementación directa, prefiriendo el uso de interfaces como alternativa.

El siguiente ejemplo aborda la implementación de la herencia múltiple aplicado a un sistema de autenticación con múltiples métodos de seguridad.

En una empresa de software, se requiere un sistema de autenticación que sustente múltiples métodos de seguridad. Un usuario puede autenticarse utilizando las credenciales tradicionales como usuario y contraseña, también puede autenticarse a través de credenciales biométricos como la huella dactilar o el reconocimiento facial. Dado que estos métodos comparten ciertas características, pero también tienen implementaciones distintas, la herencia múltiple es una solución adecuada para modelar el problema, permitiendo que una clase de usuario pueda heredar de múltiples métodos de autenticación.

El siguiente diagrama de clases UML muestra la idea de las relaciones entre clases e interfaces implementando la herencia múltiple (Figura 1.12).

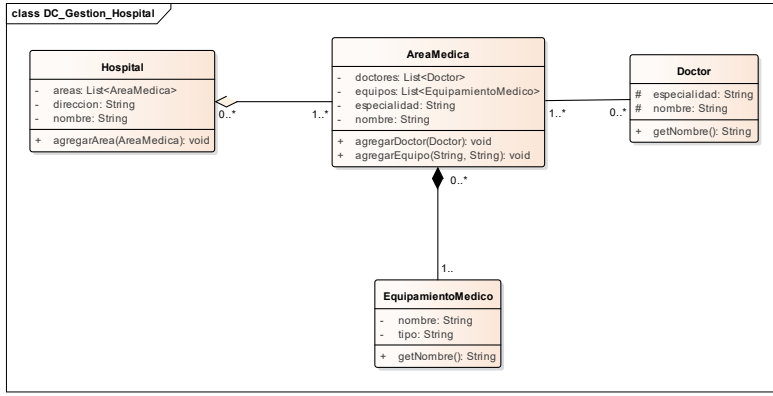


Figura 1.12. Representación UML de la herencia múltiple del ejemplo Sistema de Autenticación.

La solución implementada en java:

```
// Definición de la clase base Usuario
class Usuario {
    protected String nombreUsuario;
    protected String correo;

    public Usuario(String nombreUsuario, String correo) {
        this.nombreUsuario = nombreUsuario;
        this.correo = correo;
    }

    public void iniciarSesion() {
        System.out.println(nombreUsuario + " ha iniciado sesión". );
    }
```

```
}

// Interfaz para autenticación con contraseña
interface AutenticacionContrasena {
    void autenticarConContrasena(String contrasena);
}

// Interfaz para autenticación biométrica
interface AutenticacionBiometrica {
    void autenticarConBiometria(String datosBiometricos);
}

// Clase UsuarioSeguro que implementa herencia múltiple a través de interfaces
class UsuarioSeguro extends Usuario implements AutenticacionContrasena, AutenticacionBiometrica {
    private String contrasenaAlmacenada;
    private String datosBiometricosAlmacenados;

    public UsuarioSeguro(String nombreUsuario, String correo, String contrasena, String datosBiometricos) {
        super(nombreUsuario, correo);
        this.contrasenaAlmacenada = contrasena;
        this.datosBiometricosAlmacenados = datosBiometricos;
```

```
    }

    @Override
    public void autenticarConContrasena(String contrasena) {
        if (this.contrasenaAlmacenada.equals(contrasena)) {
            System.out.println("Autenticación por contraseña exitosa para " + nombreUsuario);
        } else {
            System.out.println("Error de autenticación por contraseña para " + nombreUsuario);
        }
    }

    @Override
    public void autenticarConBiometria(String datosBiometricos) {
        if (this.datosBiometricosAlmacenados.equals(datosBiometricos)) {
            System.out.println("Autenticación biométrica exitosa para " + nombreUsuario);
        } else {
            System.out.println("Error de autenticación biométrica para " + nombreUsuario);
        }
    }
}
```

```
// Clase principal para pruebas
public class SistemaAutenticacion {
    public static void main(String[] args) {
        UsuarioSeguro usuario = new UsuarioSeguro("Carlos",
"carlos@example.com", "contrasenaSegura123", "huella-
Dactilar123");

        usuario.iniciarSesion();
        usuario.autenticarConContrasena("contrasenaSegu-
ra123");
        usuario.autenticarConBiometria("huellaDactilar123");
    }
}
```

La implementación anterior en el lenguaje java, desarrolla un enfoque híbrido de herencia simple utilizando como elemento del lenguaje para representar herencia el extends Usuario junto con implementación múltiple de interfaces utilizando el *implements AutenticacionContrasena e implements AutenticacionBiometrica*, lo cual es una solución adecuada por varias razones:

- Separa las responsabilidades: la clase Usuario maneja la funcionalidad básica de iniciar sesión con los atributos nombre usuario y correo, las interfaces se encargan de separar los diferentes métodos de autenticación y cada interfaz representa una capacidad específica de autenticación.

- Tiene gran flexibilidad para agregar nuevos métodos de autenticación sin afectar el código actual sino agregar más código reutilizando el modelo desarrollado.
- Permite que diferentes tipos de usuarios implementen solo los métodos de autenticación que necesiten.

Esta implementación evita ambigüedades y conflictos en la resolución de métodos con el mismo nombre en distintas clases padre. Las interfaces permiten definir funcionalidades específicas sin forzar una jerarquía rígida.

Si no se implementara de esta forma, podrían desarrollarse las siguientes malas soluciones:

- Uso sólo de herencia simple:

```
class UsuarioConContrasena extends Usuario {
    // Métodos de autenticación por contraseña
}
class UsuarioConBiometria extends UsuarioConContrasena {
    // Métodos de autenticación biométrica
}
```

Como se puede observar hay una jerarquía en profundidad rígida y poco flexible, la cual contribuye a heredar funcionalidades innecesarias presentando dificultades a la hora de combinar diferentes tipos de autenticación. Esto muestra un alto acoplamiento entre las clases.

- Uso de composición sin interfaces:

```
class UsuarioSeguro {
    private Usuario usuario;
```

```
    private ManejadorContrasena manejadorContrasena;
    private ManejadorBiometrico manejadorBiometrico;
}
```

En este desarrollo existe mayor complejidad en la delegación de métodos, hay una pérdida del polimorfismo que proporcionan las interfaces.

- Intento de usar herencia múltiple pura que no soportada en Java:

```
class UsuarioSeguro extends Usuario, AutenticacionContrasena, AutenticacionBiometrica {
    // Implementación
}
```

Tiene ambigüedad en la resolución de métodos acentuando el problema del diamante que consiste que en las clases padre tienen métodos con el mismo nombre solapando las intenciones reales del modelo al tratar de heredar a la subclase. Esto dificulta y complejiza el mantenimiento del código.

1.8. Polimorfismo

El polimorfismo es un principio fundamental de la POO, el cual permite que un mismo método pueda ejecutarse de diferentes maneras según el contexto. Se manifiesta a través de la sobrecarga y la sobrescritura de métodos, facilitando la reutilización del código y promoviendo el diseño flexible y mantenible de software. Su aplicación en interfaces permite una mayor abstracción y modularidad, haciendo que las clases sean más reutilizables e independientes de implementaciones específicas.

Existen algunos tipos de polimorfismos a la hora de utilizarlo, del cual se darán ejemplos de su implementación.

- Polimorfismo por Herencia teniendo en cuenta los distintos métodos con el mismo nombre de las subclases):

```
// Ejemplo: Sistema de Notificaciones de una Red Social
abstract class Notificacion {
    protected String mensaje;

    public abstract void mostrar();
}

class NotificacionComentario extends Notificacion {
    private String postId;

    @Override
    public void mostrar() {
        System.out.println("Nuevo comentario en tu post: " +
mensaje);
    }
}

class NotificacionMensaje extends Notificacion {
    private String remitente;
```

```
    @Override
    public void mostrar() {
        System.out.println("Mensaje de " + remitente + ": " +
mensaje);
    }
}

// Uso polimórfico
class GestorNotificaciones {
    public void procesarNotificacion(Notificacion notif) {
        // El mismo método maneja diferentes tipos de notifi-
caciones
        notif.mostrar();
    }
}
```

- Polimorfismo por Interfaces:

```
// Ejemplo: Sistema de Pagos en un E-commerce
interface Pagable {
    double calcularTotal();
    void procesarPago();
}

class CompraProducto implements Pagable {
    private List<Producto> productos;
```

```
    @Override
    public double calcularTotal() {
        return productos.stream()
                .mapToDouble(Producto::getPrecio)
                .sum();
    }

    @Override
    public void procesarPago() {
      System.out.println("Procesando pago de productos..". );
    }
}

class ServicioSuscripcion implements Pagable {
    private double precioMensual;
    private int meses;

    @Override
    public double calcularTotal() {
        return precioMensual * meses;
    }

    @Override
```

```
    public void procesarPago() {
        System.out.println("Procesando pago de suscripción..".
);
    }
}
```


El uso del polimorfismo en los dos casos anteriores se evidencia a través del uso de clases abstractas o interfaces, tienen métodos @Override, pasas objetos por su tipo base/interfaz y un mismo método puede manejar diferentes tipos de objetos.

- Polimorfismo por Sobrecarga (en tiempo de compilación):

```
// Ejemplo: Calculadora de Distancias
class CalculadoraDistancia {
    // Entre dos puntos 2D
    public double calcularDistancia(double x1, double y1,
double x2, double y2) {
        return Math.sqrt(Math.pow(x2-x1, 2) + Math.pow(y2-y1,
2));
    }

    // Entre dos puntos 3D
    public double calcularDistancia(double x1, double y1,
double z1,
                                    double x2, double y2, double z2) {
        return Math.sqrt(Math.pow(x2-x1, 2) +
                         Math.pow(y2-y1, 2) +
```

```
                Math.pow(z2-z1, 2));
    }

    // Entre dos ciudades por nombre
    public double calcularDistancia(String ciudad1, String
ciudad2) {
        // Lógica para obtener coordenadas de ciudades y
calcular
        return 0.0;
    }
}
```

En este caso hay varios métodos con el mismo nombre pero diferentes parámetros, en este caso el propio compilador elige el método según los argumentos.

- Polimorfismo paramétrico con genericidad:

```
// Ejemplo: Procesador de Datos Científicos
class ProcesadorDatos<T extends Number> {
    private List<T> datos;

    public double calcularPromedio() {
        return datos.stream()
                .mapToDouble(Number::doubleValue)
                .average()
                .orElse(0.0);
    }
```

```
    public void agregarDato(T dato) {
        datos.add(dato);
    }
}
// Uso
ProcesadorDatos<Integer> procesadorEnteros = new ProcesadorDatos<>();
ProcesadorDatos<Double> procesadorDecimales = new ProcesadorDatos<>();
```

En este caso se usa <T> como muestra de una genericidad, concepto que se verá en la próxima sección, en declaraciones de clase/método. Esto quiere decir que la misma clase/método funciona con diferentes tipos de datos.

- Colecciones polimórficas: es un arreglo, lista, matriz u otra estructura que almacena en tiempo de ejecución, objetos de diferentes clases que comparten una superclase o interfaz común.

```
// Sistema de gestión de empleados en una empresa
abstract class Empleado {
    protected String nombre;
    protected double salarioBase;

    public abstract double calcularSalario();
    public abstract String obtenerRol();
}
```

```
class Desarrollador extends Empleado {
    private int horasExtra;

    @Override
    public double calcularSalario() {
        return salarioBase + (horasExtra * 20);
    }

    @Override
    public String obtenerRol() {
        return "Desarrollador";
    }
}

class Gerente extends Empleado {
    private double bono;

    @Override
    public double calcularSalario() {
        return salarioBase + bono;
    }

    @Override
```

```
    public String obtenerRol() {
        return "Gerente";
    }
}

// Uso de arreglo polimórfico
class EmpresaGestion {
    public static void main(String[] args) {
        // Creación de arreglo polimórfico
        Empleado[] empleados = new Empleado[5];

        // Llenado con diferentes tipos de empleados
        empleados[0] = new Desarrollador();
        empleados[1] = new Gerente();
        empleados[2] = new Desarrollador();
        empleados[3] = new Gerente();
        empleados[4] = new Desarrollador();

        // Procesamiento polimórfico
        double totalSalarios = 0;
        for(Empleado emp : empleados) {
            totalSalarios += emp.calcularSalario();
            System.out.println("Rol: " + emp.obtenerRol());
        }
```

```
    }
}
```

El arreglo polimórfico en este caso permite manejar todos los tipos de empleados de manera uniforme, reduciendo la duplicación de código y permitiendo una gestión más flexible de los empleados

1.9. Genericidad

La genericidad en la POO permite definir clases, interfaces y métodos con parámetros donde no es necesario especificar un tipo de datos. Esto crea clases, métodos y estructuras reutilizables para cualquier tipo de datos proporcionando mayor flexibilidad, reutilización de código y seguridad en tiempo de compilación. Además, evita conversiones de tipos explícitas (casting), reduciendo errores en tiempo de ejecución. En Java, este concepto se utiliza con la sintaxis <T> (o cualquier otro identificador) y permiten manipular diferentes tipos de datos sin perder seguridad de tipos.

El siguiente ejemplo describe un sistema de gestión de artistas en un circo, utilizando una clase genérica para manejar los diferentes tipos de artistas. Para ello se debe tener en cuenta la clase genérica *ManejadorArtistas<T>* es quien se encarga de administrar los diferentes tipos de artistas. Existen clases concretas como *Payaso, Malabarista y Acróbata* las cuales constituyen especializaciones de la clase Artista que debe ser abstracta debido a que cada tipo de artista definido tendrá un acto diferente que cada subclase implementará. Para poder gestionar todos los artistas, se implementa la clase Circo quien utiliza la clase *ManejadorArtistas<T>* para gestionar artistas.

El siguiente modelo UML (Figura 1.13) describe el problema que se está analizando:

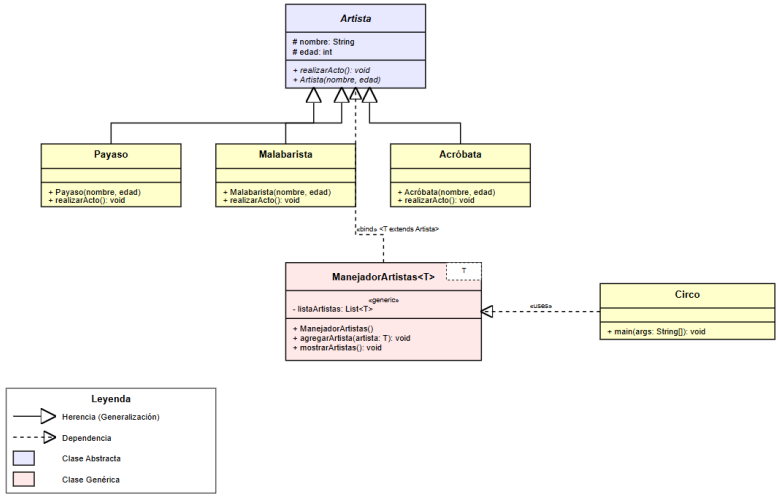


Figura 1.13. Diagrama de clases-Sistema de Gestión de Artistas del Circo.

Explicación técnica del diagrama UML:

El diagrama de clases UML representa un sistema de gestión de artistas implementando genericidad en Java, donde la arquitectura combina herencia polimórfica con contenedores parametrizados type-safe. La clase abstracta Artista, renderizada en color azul claro con nomenclatura en cursiva según convención UML para entidades no instanciables, constituye la raíz de la jerarquía de tipos y define el contrato mediante el método abstracto realizarActo() que cada especialización concreta debe implementar. Los atributos protegidos nombre y edad se heredan a las subclases mediante el mecanismo de visibilidad protected, denotado por el símbolo numeral en la notación UML.

Las clases concretas Payaso, Malabarista y Acróbata, representadas en amarillo claro, especializan la abstracción base mediante relaciones de generalización indicadas por flechas con punta

triangular blanca dirigidas hacia la superclase. Cada subclase proporciona una implementación específica del método realizarActo(), materializando el polimorfismo de subtipos, e invoca el constructor de la superclase mediante super() para garantizar la correcta inicialización de la cadena de herencia.

El elemento central del diseño es la clase genérica *ManejadorArtistas<T>*, distinguida cromáticamente en rosa claro y anotada con el estereotipo «generic» entre guillemets franceses. La genericidad se representa mediante un rectángulo punteado en la esquina superior derecha conteniendo el parámetro de tipo T, siguiendo la notación UML para template parameter boxes. El nombre de la clase incluye explícitamente la sintaxis de corchetes angulares *ManejadorArtistas<T>*, documentando formalmente su naturaleza parametrizada.

La característica técnica más significativa es el bounded type parameter, expresado mediante una relación de dependencia punteada desde ManejadorArtistas hacia Artista, anotada con el estereotipo «bind» y la restricción <T extends Artista>. Esta notación UML estándar especifica que el parámetro de tipo T está acotado superiormente por Artista, estableciendo que T debe ser Artista o cualquier subtipo de su jerarquía. Esta restricción genérica garantiza type safety en tiempo de compilación, permitiendo al compilador verificar que solo objetos compatibles con la jerarquía de Artista pueden agregarse al contenedor, eliminando necesidad de casting explícito y previniendo ClassCastException en runtime. El atributo listaArtistas de tipo List<T> y el método agregarArtista(artista: T) demuestran la propagación del parámetro de tipo a través de la estructura de la clase genérica.

La clase Circo actúa como cliente del sistema, conectada mediante dependencia punteada con

estereotipo «uses», indicando que utiliza los servicios de ManejadorArtistas. Aunque el diagrama de clases muestra la estructura genérica, la implementación concreta en el método main() realiza el binding de tipos, creando instancias como ManejadorArtistas<Payaso>, ManejadorArtistas<Malabarista> y ManejadorArtistas<Acróbata>, donde el parámetro T se sustituye por tipos concretos específicos.

Este diseño arquitectónico demuestra la combinación sinérgica de polimorfismo de subtipos mediante herencia y polimorfismo paramétrico mediante genericidad, proporcionando reutilización de código sin sacrificar seguridad de tipos. La representación UML aplica rigurosamente las especificaciones estándar para clases abstractas, relaciones de generalización, dependencias, estereotipos y parámetros de tipo, constituyendo un modelo completo y formalmente correcto para documentación técnica de arquitecturas de software que implementan patrones genéricos.

Las clases en Java

```
// Clase base para todos los artistas del circo
abstract class Artista {
    protected String nombre;
    protected int edad;

    public Artista(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
```

```
    public abstract void realizarActo();

}
```

En las siguientes clases específicas que serían los subtipos de la clase Artista, cada subclase sobrescribe el método realizarActo() con su propia implementación. Además, usan el constructor de Artista para inicializar los atributos.

```
// Clase específica Payaso
class Payaso extends Artista {
    public Payaso(String nombre, int edad) {
        super(nombre, edad);
    }

    @Override
    public void realizarActo() {
        System.out.println(nombre + " cuenta chistes y hace
trucos graciosos". );
    }
}

// Clase específica Malabarista
class Malabarista extends Artista {
    public Malabarista(String nombre, int edad) {
        super(nombre, edad);
    }
```

```
    @Override
    public void realizarActo() {
        System.out.println(nombre + " hace malabares con fuego". );
    }
}

// Clase específica Acróbata
class Acróbata extends Artista {
    public Acróbata(String nombre, int edad) {
        super(nombre, edad);
    }

    @Override
    public void realizarActo() {
        System.out.println(nombre + " realiza saltos y acrobacias en el aire". );
    }
}
```

En cuanto a la clase genérica que se encarga de manejar a cualquier tipo de artista solo acepta tipos que hereden de Artista (T extends Artista). Implementa una lista genérica que almacena artistas de diferentes tipos.

```
// Clase genérica para manejar cualquier tipo de artista
class ManejadorArtistas<T extends Artista> {
```

```
    private List<T> listaArtistas;

    public ManejadorArtistas() {
        this.listaArtistas = new ArrayList<>();
    }

    public void agregarArtista(T artista) {
        listaArtistas.add(artista);
    }

    public void mostrarArtistas() {
        for (T artista : listaArtistas) {
            System.out.println("Artista: " + artista.nombre + ",
Edad: " + artista.edad);
            artista.realizarActo();
        }
    }
}
```

Un ejemplo específico que usa esta clase genérica es a través del siguiente programa principal donde se puede visualizar la creación de tres instancias de *ManejadorArtistas<T>*, una para cada tipo de artista.

```
// Clase principal que usa la clase genérica
public class Circo {
    public static void main(String[] args) {
```

```
        // Crear manejadores para cada tipo de artista
       ManejadorArtistas<Payaso> manejadorPayasos = new
ManejadorArtistas<>();
        ManejadorArtistas<Malabarista> manejadorMalabaris-
tas = new ManejadorArtistas<>();
         ManejadorArtistas<Acróbata> manejadorAcróbatas =
new ManejadorArtistas<>();


        // Crear artistas y agregarlos a sus respectivos mane-
jadores
         manejadorPayasos.agregarArtista(new Payaso("Pepi-
to", 35));
        manejadorMalabaristas.agregarArtista(new Malabaris-
ta("Jorge", 28));
                manejadorAcróbatas.agregarArtista(new  Acróba-
ta("Laura", 24));


        // Mostrar los artistas y sus actos
        System.out.println("Circo - Lista de artistas");
        manejadorPayasos.mostrarArtistas();
        manejadorMalabaristas.mostrarArtistas();
        manejadorAcróbatas.mostrarArtistas();
    }
}
```

1.10. Cohesión y acoplamiento en la Programación Orientada a Objetos

Esto dos conceptos suelen ser de gran importancia para poder dividir adecuadamente un problema, y que su solución sea más fácil y entendible de implementar. También influyen directamente en la calidad y mantenibilidad del software. Los lineamientos que se emplean para el desarrollo de software orientado a objetos es mantener una alta cohesión y un bajo acoplamiento. Según Priolo (2009) la cohesión es la forma en que se agrupan pequeñas unidades de software (módulo, clase, librerías) en unidades mayores, y el acoplamiento por su parte indica el grado de dependencia entre estas unidades de software agrupadas.

Para diseñar clases con alta cohesión, hay que tener en cuenta, Vargas Ortega (2021)a latent problem is the lack of trained personnel who meet the necessary requirements for development processes, because recently graduated professionals, in most cases, have not acquired the necessary skills in handling the latest tools. technology and good practices. A complex software development process requires not only trained personnel, but also a high level of creativity that allows them to learn new technologies to apply them directly to projects. Training and education depends in part on the higher education institution where the individual forms their skills, and the level of creativity is acquired by actively participating in development processes where they have the opportunity to interact with real cases. Therefore, a project is proposed that allows: Carrying out a study of the design patterns suggested by GRASP for their dissemination and use in object-oriented programming, through documentary instruments that accompany a software prototype. For this reason, the undergraduate student will be able to

count on a documentary tool about the use of GRAPS design patterns, with which the paradigm of information flow diagrams is broken and a new one is created, which allows the designer to concentrate on the specific functions rather than the interaction between them, which affects the management of fundamental concepts of analysis, design and programming under the object-oriented paradigm (in software engineering processes considera que este tipo de clases posee una importante funcionalidad y poco trabajo, colabora con otros objetos para compartir tareas muy grandes. Esto significa que una clase realiza una única tarea o responsabilidad, donde sus métodos y atributos se centran en una funcionalidad específica. También el mismo autor define que el bajo acoplamiento debe tener poca dependencia entre las clases con suficiente información para que su relación con otras se reduzca y puedan en algún momento subsistir si una de las clases a las cuales se relaciona deja de existir. Lo que quiere decir que las unidades de software funcionan independientes permitiendo modificar una clase sin afectar a otras.

La cohesión puede ver reflejada visualmente a través de un diseño de clases en UML, teniendo en cuenta su definición particular, mientras que el acoplamiento se puede observar en la cantidad y tipo de relaciones entre las clases. Una clase con alto acoplamiento mantiene muchas asociaciones directas con otras clases, mientras que una clase con bajo acoplamiento utilizará interfaces o abstracciones para interactuar con otras clases.

Para ayudar a entender estos conceptos, se presentan los siguientes casos que analizan la cohesión y el acoplamiento teniendo en cuenta su intensidad.

Diseño con baja cohesión y alto acoplamiento (No Adecuado)

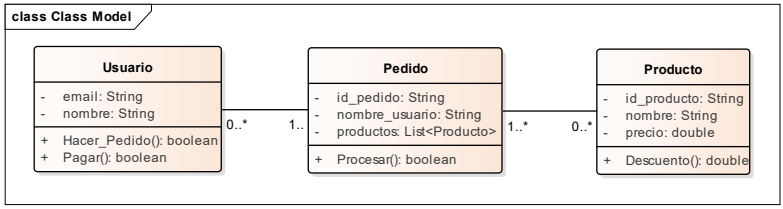


Figura 1.14. Representación UML de clases con Baja Cohesión y Alto Acoplamiento.

Este diseño (Figura 1.14) tiene clases con múltiples responsabilidades y dependencias fuertes entre ellas. La clase Usuario maneja la información del usuario y además la lógica de pagos, lo cual viola el principio de única responsabilidad. La clase Pedido maneja directamente a Usuario y Producto, lo que hace que se mezcle la lógica de negocio con la gestión de objetos. Esto describe una baja cohesión entre las clases. Al mismo tiempo visualiza un alto acoplamiento ya que la clase Usuario está relacionado directamente con la clase Pedido, lo que significa que cualquier cambio en la clase Pedido podría afectar a la clase Usuario. Y al mismo tiempo la clase Pedido depende de la clase Producto, lo que impide reutilizar Producto sin afectar a Pedido. Este tipo de diseño puede ocasionar conflictos pues establece una mezcla de responsabilidades y crea dependencias innecesarias, lo que hace difícil su mantenimiento.

Diseño con alta cohesión y bajo acoplamiento (Adecuado)

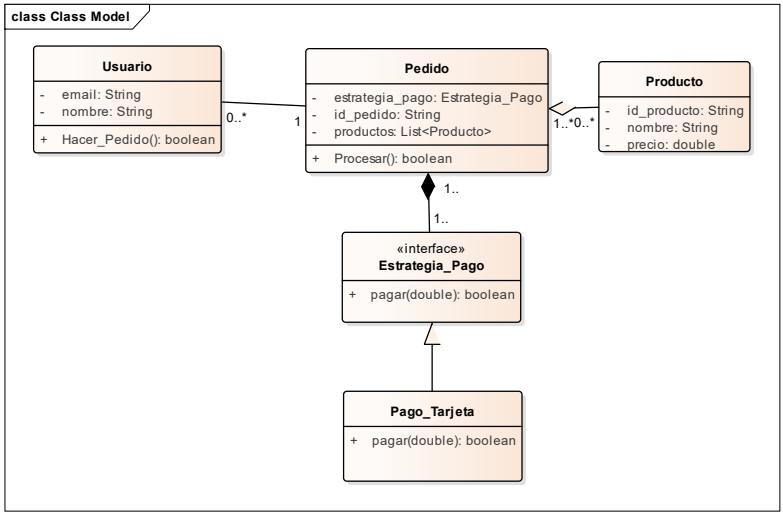


Figura 1.15. Representación UML de clases con Alta Cohesión y Bajo Acoplamiento.

En el diagrama de clases anterior (Figura 1.15), existe una clara asociación entre Usuario y Pedido, no es una relación fuerte, ya que un usuario puede hacer múltiples pedidos y estos pueden existir sin que el usuario esté activo en el sistema. También se define una agregación entre Pedido y Producto, también es una relación débil ya que la vida del producto no depende del pedido, pues los productos existen en el sistema antes y después de que se cree un pedido. Se adiciona la Interfaz Estrategia_Pago para realizar pagos flexibles y se cuenta con la clase PagoTarjeta que implementa la interfaz Estrategia_Pago, con el propósito de dejar abierto para agregar nuevos métodos como PagoPayPal, PagoTransferencia, etc., sin modificar las clases existentes. Existe una relación de composición entre la clase Estrategia_Pago y Pedido esto quiere decir que cualquier pedido tiene una estrategia de pago.

Este diseño garantiza la alta cohesión debido a que cada clase tiene una única responsabilidad como:

- La clase Usuario maneja información del usuario.
- La clase Pedido administra pedidos y su lista de productos.
- La clase Producto solo representa un producto.
- La clase Estrategia_Pago maneja métodos de pago de forma extensible.

También garantiza el bajo acoplamiento teniendo en cuenta:

- Un Usuario no depende de Pedido, lo que facilita modificaciones sin afectar otras clases.
- El Pedido y el Producto están conectados mediante agregación, lo que permite reusar productos en distintos pedidos.
- La clase Estrategia_Pago usa el patrón de estrategia, permitiendo cambiar o agregar métodos de pago sin afectar Pedido o Usuario.

1.11. Ejercicio integrador "Sistema de Gestión Académica Simplificada

Este ejercicio integrador consolida los conceptos fundamentales tratados en el capítulo mediante un único caso de estudio coherente. El objetivo no es solo "hacer que compile", sino que el estudiante sea capaz de modelar correctamente antes de programar: identificar entidades, responsabilidades, relaciones entre clases y mecanismos de extensión (herencia e interfaces). En la práctica profesional, el diseño orientado a objetos fracasa cuando se implementa desde la improvisación, por lo que aquí el proceso se plantea como una

secuencia metodológica: comprender → modelar → justificar → implementar → comprobar.

Además, el caso se diseñó para que el estudiante practique decisiones típicas:

- cuándo una relación es asociación (colaboración sin propiedad),
- cuándo es agregación (pertenencia débil),
- cuándo es composición (dependencia de ciclo de vida),
- cuándo conviene generalizar (herencia) y cuándo conviene usar un contrato (interfaces),
- y cómo mantener alta cohesión / bajo acoplamiento desde el inicio.

Caso de estudio: descripción del dominio

Una universidad necesita un sistema simplificado para gestionar el proceso académico básico de un período:

- La universidad oferta asignaturas (Ej: Programación I, Bases de Datos).
- Cada asignatura se oferta en paralelos en un período académico (Ej: P1, P2).
- Cada paralelo tiene un docente responsable y una capacidad máxima (cupo).
- Los estudiantes pueden matricularse en paralelos mientras haya cupo.
- Una matrícula registra el estado (ACTIVA / RETIRADA) y puede almacenar un conjunto de evaluaciones (parcial, final, proyecto).

- Cada evaluación genera una calificación numérica.
- La universidad también requiere notificar eventos académicos simples (por ejemplo: “matrícula registrada”, “cupo agotado”). Estas notificaciones pueden enviarse por distintos medios (correo, consola), por lo que el sistema debe permitir intercambiar el tipo de notificador sin reescribir el dominio.

Requisitos funcionales mínimos (lo que debe poder hacer el sistema):

- Registrar estudiantes y docentes.
- Crear asignaturas y ofertar paralelos con cupo.
- Matricular un estudiante en un paralelo (si hay cupo).
- Registrar evaluaciones dentro de una matrícula.
- Calcular el promedio final de una matrícula.
- Notificar eventos del proceso usando un contrato común (interfaz).

Modelado UML propuesto y explicación

Clases principales e identidad

- Persona (abstracta): representa identidad común (nombre, identificación).
- Estudiante y Docente: especializaciones claras de Persona.
- Justificación: “Estudiante es una Persona” y “Docente es una Persona”. La herencia aquí es natural, estable y evita duplicación.

Estructura académica

- Asignatura: entidad académica (código, nombre, créditos).
- Paralelo: oferta concreta de una asignatura en un período (sección, cupo, docente).

  Justificación: Paralelo no es lo mismo que Asignatura; Paralelo es la "oferta" contextual (docente + cupo + período).

Proceso (matrícula)

- Matricula: relación formal estudiante–paralelo con estado y evaluaciones.

  Justificación: aunque un estudiante "tiene matrículas", la matrícula es una entidad del dominio (no solo un vínculo). Además, contiene evaluaciones.

Evaluación

- Evaluación: componente interno de una matrícula (nombre, porcentaje, nota).

  Justificación: una evaluación no tiene sentido sin una matrícula. Si una matrícula se elimina, sus evaluaciones deben desaparecer: esto es composición.

Notificación

- Notificador (interfaz) con implementaciones NotificadorConsola y NotificadorEmailSimulado.

  Justificación: consola y email no forman una jerarquía "natural"; comparten la capacidad de notificar. Por eso se modela como interfaz.

Relaciones UML y su justificación (decisión de diseño)

1. Herencia (Generalización)

- Persona ⟶ Estudiante
- Persona ⟶ Docente

Por qué aquí sí: la identidad de ambos comparte estructura y comportamiento (nombre, id), y es estable.

2. Agregación

- Asignatura ◊── Paralelo

Por qué agregación: un paralelo depende conceptualmente de una asignatura, pero la asignatura existe como entidad académica independientemente. El paralelo puede "terminar" (por período) sin que la asignatura desaparezca.

3. Asociación

- Paralelo ── Docente

Por qué asociación: el docente existe sin el paralelo; un docente puede dictar varios paralelos.

4. Asociación con entidad intermedia

- Estudiante ── Matricula ── Paralelo

Por qué no una asociación directa: porque la matrícula tiene atributos propios (estado, fecha, evaluaciones, promedio). Esto obliga a modelarla como clase.

5. Composición

- Matricula ◆── Evaluación

Por qué composición: la evaluación es parte constitutiva de la matrícula (ciclo de vida dependiente).

6. Interfaces

- Notificador implementado por varias clases

Por qué interfaz: capacidad compartida sin jerarquía, y permite reemplazar implementaciones sin afectar el dominio.

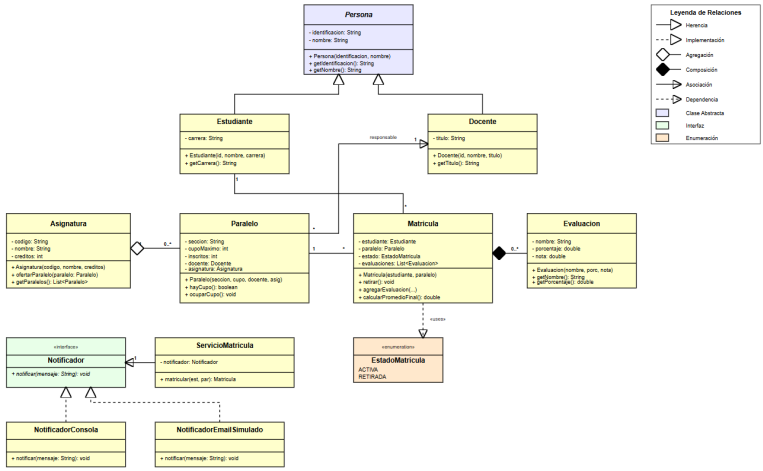


Figura 1.16. UML del Sistema de Gestión Académica Universitaria.

El diagrama de clases UML (Figura 1.16) modela un sistema de gestión académica universitaria que integra múltiples tipos de relaciones orientadas a objetos para representar la estructura organizacional, los procesos de matrícula y los mecanismos de notificación. La arquitectura implementa herencia mediante la clase abstracta Persona, renderizada en azul claro con nomenclatura en cursiva según convención UML para entidades no instanciables, que define los atributos comunes identificación y nombre con visibilidad privada. Las especializaciones concretas Estudiante y Docente, representadas en amarillo claro, extienden

esta abstracción mediante relaciones de generalización indicadas por flechas con punta triangular blanca, donde Estudiante añade el atributo carrera y Docente incorpora título, ambas clases invocando el constructor de la superclase mediante super() para garantizar la correcta inicialización de la cadena de herencia.

La estructura académica se modela mediante Asignatura y Paralelo, relacionadas por agregación denotada con un diamante blanco en el extremo de Asignatura. Esta relación semántica indica que un paralelo depende conceptualmente de una asignatura para su existencia, pero la asignatura mantiene identidad independiente y puede persistir, aunque sus paralelos finalicen al terminar un período académico. La multiplicidad 1 a 0..* especifica que una asignatura puede ofertar múltiples paralelos, mientras que cada paralelo pertenece a exactamente una asignatura. La clase Asignatura encapsula los atributos codigo, nombre y creditos, manteniendo una colección de paralelos mediante el método ofertarParalelo(), mientras que Paralelo gestiona seccion, cupoMaximo e inscritos, implementando la lógica de control de cupos mediante hayCupo() y ocuparCupo().

La relación entre Paralelo y Docente se establece mediante asociación navegable simple, representada por una línea sólida con flecha direccional, indicando que cada paralelo tiene un docente responsable pero el docente puede estar asociado a múltiples paralelos. La multiplicidad * a 1 documenta esta cardinalidad, y el rol “responsable” clarifica la naturaleza semántica de la asociación. Esta relación difiere de la herencia y la agregación porque representa una colaboración donde ambas entidades mantienen ciclos de vida completamente independientes.

El proceso de matrícula se modela mediante la clase Matricula como entidad intermedia que materializa la

relación many-to-many entre Estudiante y Paralelo. Esta decisión de diseño surge de la necesidad de asociar atributos y comportamiento al vínculo mismo, específicamente el estado de tipo EstadoMatricula (enumeración con valores ACTIVA y RETIRADA) y una colección de evaluaciones. La multiplicidad documenta que un estudiante puede tener múltiples matrículas y un paralelo puede contener múltiples matrículas, estableciendo bidireccionalidad mediante asociaciones simples desde ambas clases hacia Matricula. La clase implementa calcularPromedioFinal() que pondera las calificaciones según los porcentajes de cada evaluación, demostrando comportamiento específico del dominio encapsulado en la entidad de asociación.

La relación de composición entre Matricula y Evaluacion se denota mediante un diamante negro sólido en el extremo de Matricula, estableciendo dependencia existencial fuerte donde las evaluaciones no pueden existir sin una matrícula contenedora y su ciclo de vida está completamente subordinado. La multiplicidad 1 a 0..* indica que una matrícula puede contener cero o más evaluaciones. Esta composición implica que al eliminar una matrícula, todas sus evaluaciones asociadas deben destruirse automáticamente, reflejando la semántica de parte-todo con responsabilidad de ciclo de vida. La clase Evaluacion encapsula nombre, porcentaje y nota, representando componentes internos sin identidad independiente fuera del contexto de la matrícula.

El patrón de notificación implementa polimorfismo de interfaz mediante Notificador, representada en verde claro con el estereotipo «interface» entre guillemets franceses. Esta abstracción define el contrato notificar(mensaje: String) que debe ser implementado por las clases concretas NotificadorConsola y NotificadorEmailSimulado.

Las relaciones de implementación se representan mediante líneas punteadas con flechas triangulares blancas, distinguiéndose visualmente de la herencia de clases para enfatizar que las interfaces establecen contratos de comportamiento sin proporcionar implementación. Esta decisión arquitectónica permite intercambiar mecanismos de notificación en tiempo de ejecución mediante inyección de dependencias, siguiendo el principio de inversión de dependencias donde ServicioMatricula depende de la abstracción Notificador en lugar de implementaciones concretas.

La clase ServicioMatricula actúa como coordinadora del proceso de matrícula, manteniendo una asociación con Notificador mediante composición donde el notificador se inyecta en el constructor. El método matricular() implementa la lógica de negocio verificando disponibilidad de cupo mediante hayCupo(), ocupando el cupo con ocuparCupo(), creando la instancia de Matricula y emitiendo notificaciones mediante la interfaz. Esta arquitectura demuestra separación de responsabilidades donde la lógica de dominio (validación de cupo, creación de matrícula) se mantiene independiente del mecanismo de notificación, permitiendo extensibilidad sin modificar código existente según el principio open-closed.

La enumeración EstadoMatricula, renderizada en naranja claro con estereotipo «enumeration», define un conjunto finito de valores ACTIVA y RETIRADA que representan el estado del proceso académico. La relación de dependencia punteada desde Matricula hacia EstadoMatricula, anotada con «uses», indica que Matricula utiliza este tipo enumerado sin establecer una relación estructural fuerte, documentando dependencia de tipo en el modelo estático.

El diagrama aplica rigurosamente las convenciones UML 2.x incluyendo símbolos de visibilidad donde

menos (-) denota private, más (+) denota public y numeral (#) denota protected. Las clases abstractas se distinguen mediante nombres en cursiva, los métodos abstractos en interfaces aparecen en cursiva, y los estereotipos proporcionan metainformación semántica. Las multiplicidades en todas las asociaciones documentan cardinalidades permitidas, y los nombres de rol como "responsable" clarifican la naturaleza de las relaciones donde la semántica podría ser ambigua. La leyenda incluida documenta sistemáticamente los símbolos de relación: herencia con triángulo blanco sólido, implementación con triángulo blanco punteado, agregación con diamante blanco, composición con diamante negro, asociación con flecha simple y dependencia con flecha punteada.

Este diseño arquitectónico demuestra la aplicación coordinada de herencia para compartir estructura común, agregación para relaciones parte-todo sin dependencia existencial fuerte, composición para contención con ciclo de vida dependiente, asociaciones para colaboraciones entre entidades independientes, interfaces para contratos de comportamiento intercambiables, y enumeraciones para conjuntos finitos de valores de dominio. La combinación sinérgica de estos mecanismos produce un modelo que balancea flexibilidad, mantenibilidad y corrección semántica, constituyendo un ejemplo paradigmático de modelado orientado a objetos para sistemas de información académica con representación UML formalmente correcta adecuada para documentación técnica empresarial.

Implementación en Java del modelo UML

El código está organizado para que el estudiante vea una implementación limpia: encapsulación, responsabilidades claras, y métodos esenciales.

(A) Abstracción y herencia: Persona, Estudiante, Docente

```
abstract class Persona {
    private String identificacion;
    private String nombre;

    public Persona(String identificacion, String nombre) {
        this.identificacion = identificacion;
        this.nombre = nombre;
    }

    public String getIdentificacion() { return identificacion; }
    public String getNombre() { return nombre; }
}

class Estudiante extends Persona {
    private String carrera;

    public Estudiante(String identificacion, String nombre,
String carrera) {
        super(identificacion, nombre);
        this.carrera = carrera;
    }

    public String getCarrera() { return carrera; }
```

```
}

class Docente extends Persona {
    private String titulo;

    public Docente(String identificacion, String nombre, String
titulo) {
        super(identificacion, nombre);
        this.titulo = titulo;
    }

    public String getTitulo() { return titulo; }
}
```

(B) Estructura académica: Asignatura y Paralelo

```
import java.util.ArrayList;
import java.util.List;

class Asignatura {
    private String codigo;
    private String nombre;
    private int creditos;

    // Agregación: una asignatura puede tener múltiples pa-
ralelos ofertados
```

```
    private List<Paralelo> paralelos = new ArrayList<>();

    public Asignatura(String codigo, String nombre, int cred-
itos) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.creditos = creditos;
    }

    public String getCodigo() { return codigo; }
    public String getNombre() { return nombre; }
    public int getCreditos() { return creditos; }

    public void ofertarParalelo(Paralelo paralelo) {
        paralelos.add(paralelo);
    }

    public List<Paralelo> getParalelos() {
        return new ArrayList<>(paralelos);
    }
}

class Paralelo {
    private String seccion;
```

```
    private int cupoMaximo;
    private Docente docente; // Asociación
    private Asignatura asignatura; // Agregación (referencia a
la asignatura)
    private int inscritos;

    public Paralelo(String seccion, int cupoMaximo, Docente
docente, Asignatura asignatura) {
        this.seccion = seccion;
        this.cupoMaximo = cupoMaximo;
        this.docente = docente;
        this.asignatura = asignatura;
        this.inscritos = 0;
    }

    public String getSeccion() { return seccion; }
    public int getCupoMaximo() { return cupoMaximo; }
    public int getInscritos() { return inscritos; }
    public Docente getDocente() { return docente; }
    public Asignatura getAsignatura() { return asignatura; }

    public boolean hayCupo() {
        return inscritos < cupoMaximo;
    }
```

```
    public void ocuparCupo() {
        if (!hayCupo()) {
            throw new IllegalStateException("No hay cupo disponible en el paralelo " + seccion);
        }
        inscritos++;
    }
}
```

(C) Proceso: Matrícula y Evaluación (composición)

```
import java.util.ArrayList;
import java.util.List;

enum EstadoMatricula {
    ACTIVA, RETIRADA
}

class Evaluacion {
    private String nombre;
    private double porcentaje; // 0 a 100
    private double nota;       // 0 a 10

    public Evaluacion(String nombre, double porcentaje, double nota) {
        this.nombre = nombre;
        this.porcentaje = porcentaje;
```

```
        this.nota = nota;
    }

    public String getNombre() { return nombre; }
    public double getPorcentaje() { return porcentaje; }
    public double getNota() { return nota; }
}

class Matricula {
    private Estudiante estudiante; // Asociación
    private Paralelo paralelo;     // Asociación
    private EstadoMatricula estado;

    // Composición: evaluaciones existen "dentro" de la matrícula
    private List<Evaluacion> evaluaciones = new ArrayList<>();

    public Matricula(Estudiante estudiante, Paralelo paralelo)
{
        this.estudiante = estudiante;
        this.paralelo = paralelo;
        this.estado = EstadoMatricula.ACTIVA;
    }

    public Estudiante getEstudiante() { return estudiante; }
```

```
    public Paralelo getParalelo() { return paralelo; }
    public EstadoMatricula getEstado() { return estado; }

    public void retirar() {
        this.estado = EstadoMatricula.RETIRADA;
    }

    public void agregarEvaluacion(String nombre, double porcentaje, double nota) {
        evaluaciones.add(new Evaluacion(nombre, porcentaje, nota));
    }

    public double calcularPromedioFinal() {
        double suma = 0;
        for (Evaluacion e : evaluaciones) {
            suma += (e.getNota() * (e.getPorcentaje() / 100.0));
        }
        return suma;
    }

    public List<Evaluacion> getEvaluaciones() {
        return new ArrayList<>(evaluaciones);
    }
}
```

(D) Interfaz: Notificador y dos implementaciones

```
interface Notificador {
    void notificar(String mensaje);
}

class NotificadorConsola implements Notificador {
    @Override
    public void notificar(String mensaje) {
        System.out.println("[NOTIFICACIÓN] " + mensaje);
    }
}

class NotificadorEmailSimulado implements Notificador {
    @Override
    public void notificar(String mensaje) {
        System.out.println("[EMAIL SIMULADO] Enviando co-
rreo: " + mensaje);
    }
}
```

(E) Servicio de caso de uso: Matricular con notificación

```
class ServicioMatricula {
    private Notificador notificador;

    public ServicioMatricula(Notificador notificador) {
```

```
        this.notificador = notificador;
    }

    public Matricula matricular(Estudiante estudiante, Paralelo paralelo) {
        if (!paralelo.hayCupo()) {
            notificador.notificar("Cupo agotado en " + paralelo.getAsignatura().getNombre()
                    + " - Paralelo " + paralelo.getSeccion());
            throw new IllegalStateException("No es posible matricular: cupo agotado". );
        }

        paralelo.ocuparCupo();
        Matricula m = new Matricula(estudiante, paralelo);

        notificador.notificar("Matrícula registrada: " + estudiante.getNombre()
                + " en " + paralelo.getAsignatura().getNombre() + " (" + paralelo.getSeccion() + ")");
        return m;
    }
}
```

(F) Demostración completa (Main)

```
public class DemoGestionAcademica {
```

```
    public static void main(String[] args) {
        Docente docente = new Docente("D-100", "Dra. Lorena
Ruiz", "PhD en Computación");
        Estudiante estudiante = new Estudiante("E-200", "Mi-
guel Pérez", "Ingeniería en Sistemas");

        Asignatura prog1 = new Asignatura("PRG101", "Pro-
gramación I", 5);
        Paralelo p1 = new Paralelo("P1", 2, docente, prog1);
        prog1.ofertarParalelo(p1);

        // Cambia aquí el tipo de notificador sin tocar el dominio:
        Notificador notificador = new NotificadorConsola();
        ServicioMatricula servicio = new ServicioMatricula(no-
tificador);

        Matricula m1 = servicio.matricular(estudiante, p1);

        m1.agregarEvaluacion("Parcial", 40, 8.5);
        m1.agregarEvaluacion("Proyecto", 30, 9.0);
        m1.agregarEvaluacion("Final", 30, 7.5);

        System.out.println("Promedio final: " + m1.calcularPro-
medioFinal());
    }
}
```

Análisis final del ejercicio

Este caso demuestra que el diseño no es decoración: define cómo crecerá el sistema. Se observa que:

- La herencia se usa solo donde hay identidad estable (Persona →Estudiante/Docente).
- La matrícula existe como entidad porque posee estado y datos propios (no es “solo una relación”).
- La evaluación es composición porque sin matrícula carece de sentido.
- La notificación se modela por interfaces porque representa una capacidad intercambiable sin jerarquía natural.

En términos de calidad:

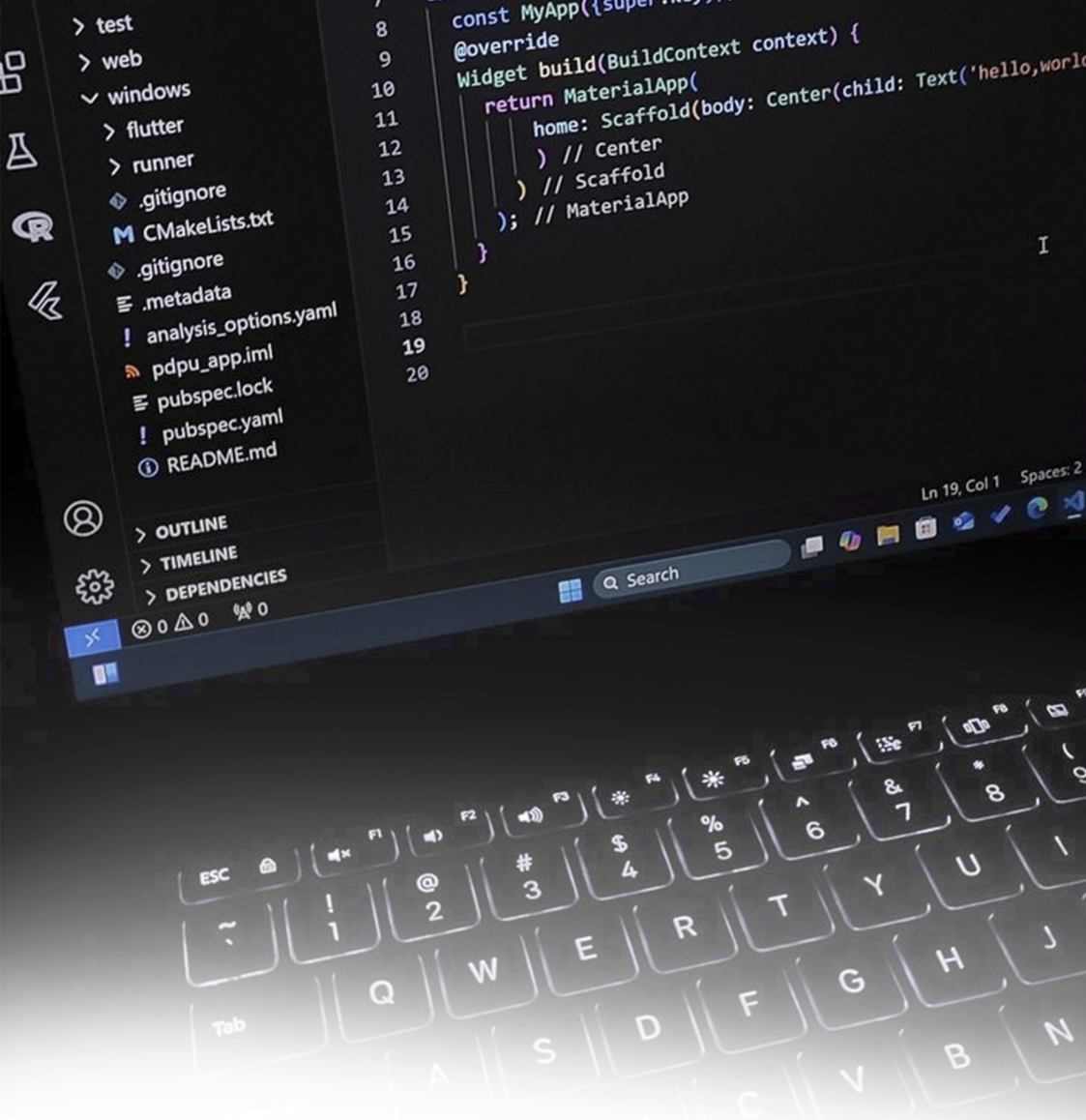
- Hay alta cohesión: cada clase tiene un propósito claro.
- Hay bajo acoplamiento: el servicio depende de Notificador, no de una implementación concreta.
- El modelo está listo para extenderse: agregar NotificadorWhatsAppSimulado, o un nuevo tipo de evaluación, o un nuevo flujo de matrícula, no obliga a reescribir el corazón del sistema.

En este primer capítulo se consolidaron los conceptos fundamentales que sostienen la Programación Orientada a Objetos como paradigma: abstracción, encapsulamiento, herencia y polimorfismo. Se argumentó su relevancia no solo como teoría, sino como fundamento práctico para la construcción de sistemas mantenibles, escalables y reutilizables. Asimismo, se introdujo UML como lenguaje estándar de modelado, destacando su papel como herramienta de comunicación técnica y de validación del diseño antes de escribir código.

Además, se profundizó en el valor de definir correctamente las relaciones entre clases (asociación, agregación y composición), entendiendo que estas decisiones no son triviales: determinan el ciclo de vida de los objetos, el grado de dependencia entre componentes y la claridad del modelo. Se incorporaron ejemplos y casos comparativos donde el contexto modifica la relación apropiada, reforzando una idea esencial: en POO no basta con conocer las definiciones; hay que justificar la decisión según el dominio.

El capítulo también estableció un puente hacia prácticas de diseño más profesionales al introducir nociones como cohesión y acoplamiento, mostrando que un sistema no solo debe "funcionar", sino estar estructurado para evolucionar. El ejercicio integrador permitió comprobar que UML y Java pueden alinearse de forma coherente cuando el análisis se realiza con disciplina, y que las decisiones de modelado se reflejan directamente en código más claro y organizado.

En el Capítulo 2, el enfoque avanzará desde los conceptos base hacia un nivel más aplicado: el estudiante comenzará a identificar errores típicos de diseño en clases, comprenderá por qué aparecen clases monolíticas ("clases Dios"), y aprenderá criterios prácticos para mejorar la estructura interna del software. En otras palabras, si el Capítulo 1 enseña a "construir con piezas correctas", el Capítulo 2 enseña a organizar esas piezas con buenas decisiones de diseño, preparando el camino para controlar el crecimiento del sistema sin perder mantenibilidad.



02

Diseño correcto de clases y responsabilidades

2.1. ¿Qué es una “buena clase”?

Uno de los errores más frecuentes en los estudiantes que comienzan a programar con orientación a objetos no es la sintaxis, ni el desconocimiento del lenguaje, sino la forma de pensar el problema. Tras aprender qué es una clase, qué es un objeto y cómo funciona la herencia, muchos estudiantes asumen que “saber POO” consiste únicamente en usar class, extends o implements. Sin embargo, la verdadera dificultad aparece cuando deben decidir cuántas clases crear, qué responsabilidad darle a cada una y cómo deben relacionarse entre sí.

Diseñar correctamente clases no es un proceso automático ni mecánico. Es un ejercicio de análisis, reflexión y toma de decisiones. Un mismo problema puede resolverse con distintos diseños, pero no todos tendrán la misma calidad, claridad ni facilidad de mantenimiento. Este capítulo tiene como objetivo enseñar al estudiante cómo pensar una solución orientada a objetos, más allá de escribir código que “funcione”.

Después de comprender los fundamentos de la POO y las relaciones entre clases tratados en el Capítulo 1, el siguiente paso natural es aprender a diseñar clases correctamente, asignar responsabilidades adecuadas y evitar estructuras confusas o sobrecargadas.

En esta sección se ha puesto énfasis en la importancia de pensar antes de programar y en comprender que la Programación Orientada a Objetos no se limita a la sintaxis de un lenguaje, sino que requiere un proceso previo de análisis y modelado. Entender el problema, identificar los conceptos relevantes y reconocer sus relaciones constituye el primer paso para construir soluciones correctas.

Sin embargo, una vez identificados estos conceptos, surge una pregunta fundamental: ¿cómo debe diseñarse

cada clase para que represente correctamente una parte del problema? No todas las clases están bien diseñadas, aunque el programa funcione. Por esta razón, en la siguiente sección se abordará qué características definen a una buena clase y cómo evaluar si una clase cumple adecuadamente su propósito dentro del sistema.

Comprender qué es una "buena clase" constituye uno de los aprendizajes más importantes dentro de la Programación Orientada a Objetos. No se trata únicamente de saber declarar una clase en un lenguaje como Java, sino de modelar correctamente una parte del problema real que se desea resolver. Una clase mal concebida puede funcionar inicialmente, pero a medida que el sistema crece se convierte en una fuente constante de errores, confusión y dificultad para realizar cambios.

Desde el punto de vista del diseño, una clase debe representar un concepto claramente identificable del dominio del problema. Dicho concepto puede corresponder a una entidad física (como Estudiante, Libro o Producto), a una entidad lógica (Pedido, Factura, Préstamo) o a un rol funcional (GestorUsuarios, ValidadorDatos). En todos los casos, la clave está en que la clase tenga sentido por sí misma y que su existencia esté justificada dentro del sistema.

Una forma práctica de evaluar si una clase está bien diseñada es analizar su **nombre**. El nombre de la clase debe ser lo suficientemente claro como para que cualquier lector del código pueda inferir su propósito sin necesidad de revisar toda su implementación.

Una buena clase no se define por la cantidad de código que contiene ni por la complejidad de sus métodos. Tampoco por la cantidad de atributos o por el uso de herencia. Una buena clase se caracteriza,

principalmente, por tener una responsabilidad clara y bien delimitada.

Desde una perspectiva conceptual, una clase representa una idea coherente del dominio del problema. Cuando se observa el nombre de la clase, debería ser relativamente sencillo responder a la pregunta:

¿De qué se encarga esta clase?

Si la respuesta es confusa, extensa o incluye varias acciones diferentes, probablemente la clase esté mal diseñada.

Características de una buena clase

Una clase bien diseñada suele cumplir con las siguientes características:

- Tiene un propósito claro.
- Representa una entidad o concepto reconocible del problema.
- Sus atributos y métodos están directamente relacionados con ese concepto.
- No realiza tareas que corresponden a otras clases.
- Puede entenderse sin necesidad de conocer todo el sistema.

Relación entre nombre, atributos y métodos

En una buena clase existe coherencia entre su nombre, sus atributos y sus métodos. Todos estos elementos deben girar en torno al mismo concepto central. Por ejemplo, si una clase se llama Estudiante, es razonable que contenga atributos como nombre, identificación, carrera o correo institucional, y métodos relacionados con acciones propias del estudiante, como actualizar

sus datos o consultar su información académica básica.

Ejemplo de clase bien definida:

```
class Estudiante {
    private String nombre;
    private String matricula;
    private String carrera;

    public Estudiante(String nombre, String matricula, String carrera) {
        this.nombre = nombre;
        this.matricula = matricula;
        this.carrera = carrera;
    }

    public String getNombre() {
        return nombre;
    }

    public String getCarrera() {
        return carrera;
    }
}
```

En este ejemplo, todos los atributos y métodos están alineados con el concepto de estudiante. La clase no

realiza operaciones que no le corresponden, ni asume responsabilidades ajenas a su naturaleza.

Por el contrario, cuando una clase comienza a incluir métodos que no guardan relación directa con su propósito principal, el diseño empieza a deteriorarse. Esto suele ocurrir cuando se intenta "resolver rápido" un problema agregando funcionalidades a una clase existente, en lugar de analizar si corresponde crear una nueva clase.

Ejemplo de una clase con problemas de diseño:

```
class Estudiante {
    private String nombre;
    private String matricula;

    public void guardarEnBaseDatos() {
        // lógica de persistencia
    }

    public void enviarCorreoNotificacion() {
        // envío de correos
    }

    public void generarReportePDF() {
        // generación de reportes
    }
}
```

Aunque el código puede funcionar, esta clase mezcla responsabilidades que no forman parte del concepto Estudiante. El resultado es una clase difícil de mantener, ya que cualquier cambio relacionado con almacenamiento, comunicación o reportes obliga a modificarla.

La clase como unidad de abstracción

En programación orientada a objetos, la clase actúa como plantilla o unidad de abstracción para definir tanto los atributos (datos) como los métodos (comportamientos) de los objetos, permitiendo modelar entidades reales de forma reusable y consistente, y facilitando el diseño conceptual y estructural de sistemas complejos (Al-Fedaghi, 2021). En consecuencia, una buena clase también cumple una función clave como unidad de abstracción. Esto significa que encapsula detalles internos y expone únicamente lo necesario para que otras partes del sistema interactúen con ella. Desde fuera de la clase, no debería ser necesario conocer cómo se implementan internamente sus operaciones, sino únicamente qué hace, no cómo lo hace.

Este principio es especialmente importante en contextos educativos, ya que ayuda al estudiante a separar el *qué* del *cómo*, fomentando una mentalidad más cercana al diseño profesional de software.

Por ejemplo, si una clase ofrece un método calcularPromedio(), el resto del sistema no necesita saber si el cálculo se realiza con un ciclo, una fórmula matemática o una estructura de datos específica. Lo único relevante es que el método cumple su función correctamente.

Preguntas guía para evaluar una clase

Antes de dar por correcta una clase, es recomendable que el estudiante se formule preguntas como las siguientes:

- ¿Puedo describir la clase en una sola frase?
- ¿Todos los métodos están relacionados con esa descripción?
- ¿Esta clase tiene más de una razón para cambiar?
- ¿Estoy usando esta clase para “resolver todo” por comodidad?
- ¿Podría dividir esta clase en dos o más clases más simples?

Responder honestamente a estas preguntas permite detectar problemas de diseño en etapas tempranas, cuando aún es fácil corregirlos.

Importancia de una buena clase en sistemas grandes

En sistemas pequeños, los errores de diseño pueden pasar desapercibidos. Sin embargo, a medida que un sistema crece, las clases mal diseñadas se convierten en un obstáculo serio para la evolución del software. Una buena clase, en cambio, facilita:

- La comprensión del sistema por parte de otros desarrolladores.
- La reutilización del código en otros proyectos.
- La corrección de errores sin efectos colaterales.
- La extensión del sistema con nuevas funcionalidades.

Por estas razones, aprender a diseñar buenas clases desde los primeros cursos de programación resulta fundamental para la formación de ingenieros y tecnólogos en sistemas.

A lo largo de esta sección se ha analizado qué se entiende por una buena clase y la importancia de que exista coherencia entre su nombre, sus atributos y sus métodos. Una clase bien diseñada representa un

concepto claro del dominio del problema y evita asumir tareas que no le corresponden.

No obstante, incluso cuando se comprende qué es una buena clase, uno de los errores más frecuentes consiste en asignar demasiadas responsabilidades a una sola clase, generalmente por comodidad o desconocimiento. Para evitar este problema, es necesario introducir un criterio fundamental en el diseño orientado a objetos: la correcta distribución de responsabilidades. Este aspecto será desarrollado en la siguiente sección, donde se analizará el principio de responsabilidad única desde una perspectiva práctica y aplicada.

2.2. Responsabilidad única (SRP)

Aunque en la literatura de ingeniería de software este concepto se asocia al Principio de Responsabilidad Única, se retoma lo definido ya en el capítulo 1, abordándole desde desde una perspectiva práctica y comprensible, sin necesidad de entrar aún en marcos teóricos formales.

Idea central: Una clase debería tener una sola responsabilidad principal.

Esto significa que una clase debe tener una razón principal para cambiar. Si una clase cambia por múltiples motivos distintos, entonces está asumiendo demasiadas responsabilidades.

Dicho de forma más simple: si una clase hace “muchas cosas diferentes”, probablemente esté mal diseñada.

¿Qué se entiende por “responsabilidad”?

En el contexto de la Programación Orientada a Objetos, una responsabilidad es una tarea o conjunto de tareas relacionadas entre sí que le corresponden naturalmente a una clase dentro del sistema. No se trata de contar

métodos, sino de analizar la naturaleza de lo que la clase hace.

Por ejemplo, en un sistema informático suelen aparecer responsabilidades como:

- Representar información (datos).
- Validar información.
- Guardar o recuperar información.
- Enviar notificaciones.
- Coordinar procesos.
- Calcular valores.

Una misma clase no debería encargarse de varias de estas responsabilidades al mismo tiempo, salvo en casos muy justificados.

Ejemplo intuitivo

Supongamos un sistema sencillo de gestión de usuarios. Un estudiante podría proponer la siguiente clase:

```
class Usuario {
    private String nombre;
    private String correo;

    public void registrarUsuario() {
        // registrar usuario
    }

    public void guardarEnArchivo() {
```

```
        // guardar datos
    }

    public void enviarCorreoBienvenida() {
        // enviar correo
    }
}
```


A primera vista, esta clase parece razonable. Sin embargo, si se analiza con cuidado, se observa que:

- Maneja datos del usuario.
- Maneja persistencia (guardar en archivo).
- Maneja comunicación (correo).

Son tres responsabilidades distintas concentradas en una sola clase.

¿Por qué este diseño es problemático?

El principal problema de este diseño no es que "no funcione", sino que **no escala bien**. Cuando se afirma que un diseño "no escala bien", no se está haciendo referencia únicamente al tamaño del sistema, sino a su capacidad de crecer, adaptarse y mantenerse comprensible a medida que aumentan los requisitos y la complejidad.

Desde el punto de vista técnico, un diseño orientado a objetos escala bien cuando puede soportar:

- La incorporación de nuevas funcionalidades.
- Cambios en las reglas de negocio.
- Aumento del número de clases, módulos o usuarios.

- Modificaciones tecnológicas (por ejemplo, cambiar la forma de almacenamiento o comunicación).

Todo esto sin que sea necesario reescribir grandes partes del sistema ni introducir errores colaterales.

Por el contrario, un diseño no escala bien cuando pequeños cambios generan efectos en cadena, obligando a modificar múltiples partes del código que no deberían verse afectadas.

Analicemos qué ocurriría en los siguientes escenarios:

- Cambia la forma de guardar los datos (de archivos a bases de datos).
- Cambia el proveedor de correo electrónico.
- Se agregan validaciones adicionales al registro.

Cada uno de estos cambios obliga a modificar la misma clase, aun cuando los cambios no están relacionados entre sí. Esto genera:

- Código difícil de mantener.
- Mayor riesgo de introducir errores.
- Clases grandes y confusas.
- Dependencias innecesarias.

Este tipo de clase suele crecer rápidamente y convertirse en lo que se conoce informalmente como una clase sobrecargada.

Cómo identificar violaciones a la responsabilidad única

Un estudiante puede aprender a detectar este problema haciéndose preguntas sencillas como:

- ¿Esta clase hace más de una cosa distinta?
- ¿Tiene métodos que no están relacionados entre sí?

- ¿Cambiaría esta clase por motivos diferentes?
- ¿Estoy agregando métodos “porque es cómodo” y no porque corresponde?

Si la respuesta es afirmativa en varios casos, es una señal clara de que la clase debe ser rediseñada.

Separación de responsabilidades: enfoque correcto

La solución no consiste en eliminar funcionalidad, sino en distribuir las responsabilidades en clases adecuadas. Cada clase debe encargarse de una tarea bien definida.

Una posible refactorización del ejemplo anterior sería:

```
class Usuario {
    private String nombre;
    private String correo;

    public Usuario(String nombre, String correo) {
        this.nombre = nombre;
        this.correo = correo;
    }

    public String getCorreo() {
        return correo;
    }
}

class RepositorioUsuario {
```

```
    public void guardar(Usuario usuario) {
        // guardar usuario en archivo o base de datos
    }
}

class ServicioCorreo {
    public void enviarBienvenida(Usuario usuario) {
        // enviar correo de bienvenida
    }
}

class GestorUsuarios {
    private RepositorioUsuario repositorio;
    private ServicioCorreo servicioCorreo;

    public GestorUsuarios(RepositorioUsuario repositorio,
ServicioCorreo servicioCorreo) {
        this.repositorio = repositorio;
        this.servicioCorreo = servicioCorreo;
    }

    public void registrarUsuario(Usuario usuario) {
        repositorio.guardar(usuario);
        servicioCorreo.enviarBienvenida(usuario);
```

```
    }
}
```

Beneficios del nuevo diseño

Con esta separación:

- Cada clase tiene **una responsabilidad clara**.
- Los cambios en persistencia no afectan la lógica de correo.
- Los cambios en correo no afectan el modelo de usuario.
- El sistema es más fácil de entender, probar y extender.

Además, el código refleja mejor la realidad del problema: el *Usuario* no "sabe" cómo se guarda ni cómo se envían correos; esas son tareas externas.

Analogía con la vida real

Para comprender mejor este concepto, pensemos en una analogía cotidiana. En una universidad:

- Un estudiante estudia.
- El departamento académico gestiona matrículas.
- El sistema de correos envía notificaciones.

No se espera que el estudiante haga todas esas tareas. De la misma forma, en un sistema informático, no se debe esperar que una sola clase lo haga todo.

La responsabilidad única no es una regla rígida, sino una guía de diseño que ayuda a crear clases claras, cohesivas y fáciles de mantener. Aprender a aplicarla desde los primeros pasos en la Programación

Orientada a Objetos permite evitar errores comunes y sentar bases sólidas para el desarrollo de software de calidad.

Se ha mostrado cómo la falta de una adecuada separación de responsabilidades conduce a clases sobrecargadas, difíciles de mantener y propensas a errores. A través de ejemplos concretos, se evidenció que una clase debe tener una única responsabilidad principal y que concentrar múltiples tareas en una sola estructura afecta negativamente la calidad del diseño.

Cuando este criterio no se aplica de manera sistemática, el resultado suele ser la aparición de clases excesivamente grandes y centralizadas, que intentan resolver la mayor parte del sistema desde un único punto. Estas clases representan uno de los problemas de diseño más comunes en la Programación Orientada a Objetos y constituyen el tema de la siguiente sección, donde se analizarán en detalle las llamadas clases Dios.

2.3. Clases “Dios”: el error más común

En el diseño orientado a objetos existe un tipo de clase que aparece con muchísima frecuencia en proyectos académicos y en desarrolladores con poca experiencia en diseño: la denominada clase Dios. También suele conocerse, en un lenguaje más descriptivo, como clase todopoderosa, clase monolítica o clase centralizada, ya que concentra una cantidad excesiva de responsabilidades dentro de una sola estructura.

Clases “Dios” (God Class)

Una clase “Dios” es un antipatrón de diseño y un code smell que ocurre cuando una sola clase asume demasiadas responsabilidades, centralizando lógica y funcionalidades que deberían distribuirse entre varias clases especializadas. Estas clases crecen en tamaño

y complejidad, violan principios de diseño orientado a objetos como el Principio de Responsabilidad Única (SRP) y conducen a acoplamientos fuertes y baja cohesión. Como resultado, el mantenimiento, la extensibilidad y la comprensión del sistema se deterioran significativamente. Tal como lo describen Haroon et al. (2025) al analizar code smells en ingeniería de software, una God Class "centraliza una cantidad excesiva de lógica y funcionalidad en una única clase, violando principios de diseño como bajo acoplamiento y alta cohesión", lo que indica un diseño pobre y potencial impacto negativo en la calidad del software.

Una clase Dios es una clase que sabe demasiado y hace demasiado. No solo representa un concepto del dominio, sino que además controla procesos, valida datos, gestiona almacenamiento, coordina acciones y, en muchos casos, interactúa directamente con el usuario. Este tipo de diseño puede parecer cómodo al inicio, pero se convierte rápidamente en uno de los principales obstáculos para la evolución del sistema.

Señales de una clase Dios

Una clase suele convertirse en una clase Dios cuando presenta una o varias de las siguientes características:

- Tiene muchos métodos que no guardan relación directa entre sí.
- Maneja simultáneamente lógica de negocio, persistencia, validaciones y comunicación.
- Es utilizada desde muchas partes del sistema, convirtiéndose en un punto central obligatorio.
- Cualquier cambio, por pequeño que sea, obliga a modificarla.

- Su tamaño crece constantemente a medida que se agregan nuevas funcionalidades.

Estas señales no suelen aparecer de forma repentina. Generalmente, la clase Dios nace como una clase aparentemente razonable, pero a medida que el sistema crece, se le van agregando responsabilidades "por comodidad", hasta que termina asumiendo tareas que no le corresponden.

Ejemplo de clase Dios

```
class SistemaAcademico {

    public void registrarEstudiante() {
        // lógica de registro
    }

    public void validarDatos() {
        // validaciones
    }

    public void guardarEstudiante() {
        // persistencia
    }

    public void asignarCurso() {
        // lógica académica
    }
```

```
    public void generarReporte() {
        // generación de reportes
    }
}
```

Esta clase concentra demasiadas responsabilidades:

- Gestión de estudiantes.
- Validaciones.
- Persistencia.
- Lógica académica.
- Reportes.

El resultado es una clase difícil de entender, mantener y extender.

Análisis detallado del problema

Para comprender por qué este diseño es incorrecto, es necesario analizar qué representa realmente esta clase. El nombre SistemaAcademico es tan genérico que no describe una responsabilidad concreta. En la práctica, esta clase actúa como:

- Un controlador del sistema.
- Un gestor de estudiantes.
- Un validador de datos.
- Un repositorio de información.
- Un generador de reportes.

Esto implica que la clase no representa un concepto claro del dominio, sino que intenta resolver todo el sistema desde un único punto. Cuando esto ocurre, el diseño pierde modularidad y claridad.

Además, cualquier cambio en una de esas áreas afecta directamente a la misma clase. Por ejemplo:

- Cambiar la forma de almacenar estudiantes (archivo → base de datos).
- Modificar reglas de validación.
- Agregar un nuevo tipo de reporte.
- Cambiar la lógica de asignación de cursos.

Todos estos cambios, aunque conceptualmente distintos, obligan a modificar la misma clase, aumentando el riesgo de errores colaterales.

Consecuencias prácticas de una clase Dios

El uso de clases Dios genera múltiples problemas técnicos y pedagógicos:

- Alta complejidad: la clase crece de forma descontrolada.
- Baja legibilidad: entender qué hace la clase requiere recorrer muchos métodos.
- Dificultad para pruebas: es complicado probar una clase que hace demasiadas cosas.
- Alto acoplamiento: otras clases dependen directamente de ella.
- Escasa reutilización: la clase no puede reutilizarse en otros contextos.

Ejemplo alternativo: clase Dios en un contexto cotidiano

Para reforzar la comprensión, consideremos otro ejemplo frecuente: un sistema de ventas.

```
class SistemaVentas {
```

```
    public void registrarProducto() { }
    public void validarStock() { }
    public void procesarPago() { }
    public void emitirFactura() { }
    public void guardarVenta() { }
    public void enviarCorreoCliente() { }
}
```


Aquí ocurre el mismo problema: una sola clase intenta manejar productos, pagos, facturación, persistencia y comunicación. Aunque el sistema funcione, su diseño es frágil y poco profesional.

Por qué los estudiantes tienden a crear clases Dios

Existen varias razones por las cuales este error es tan común:

- Se intenta “resolver rápido” el problema.
- Se piensa en el programa como un todo, no como partes.
- Se confunde centralización con organización.
- Falta experiencia en dividir responsabilidades.
- Se prioriza que el código funcione sobre que esté bien diseñado.

Reconocer estas causas es importante, ya que permite al estudiante corregir su forma de pensar antes de que el problema se vuelva estructural.

Las clases Dios no aparecen por casualidad, son el resultado directo de ignorar la correcta distribución de responsabilidades y de no aplicar criterios de cohesión y separación de tareas. Para solucionar este

problema, no basta con identificarlo; es necesario aprender a descomponer una clase grande en clases más pequeñas y coherentes, proceso que se abordará en el siguiente apartado mediante técnicas de refactorización y análisis de responsabilidades.

2.4. Refactorización: de mal diseño a buen diseño

La refactorización es una práctica fundamental de la ingeniería de software orientada a mejorar la estructura interna del código y del diseño, sin alterar el comportamiento externo del sistema, con el objetivo de incrementar su mantenibilidad, legibilidad y calidad evolutiva. Diversos estudios coinciden en que la refactorización constituye un mecanismo clave para reducir complejidad estructural y gestionar la deuda técnica en sistemas orientados a objetos, especialmente durante las fases de mantenimiento y evolución del software (Abid et al., 2020; Akhtar et al., 2022; Khaleel y Mahmood, 2024)there has been a dramatic increase and industry demand for tools and techniques on software refactoring in the last ten years, defined traditionally as a set of program transformations intended to improve the system design while preserving the behavior. Refactoring studies are expanded beyond code-level restructuring to be applied at different levels (architecture, model, requirements, etc..

Identificar una clase Dios es un avance importante, pero no suficiente. En la práctica profesional y académica, el verdadero aprendizaje ocurre cuando se es capaz de corregir un mal diseño sin alterar el comportamiento del sistema. A este proceso se le conoce como refactorización.

Refactorizar no significa “rehacer todo desde cero”, sino mejorar la estructura interna del código, reorganizando clases y responsabilidades, de manera que el sistema

sea más claro, modular y fácil de mantener, conservando exactamente la misma funcionalidad externa.

En esta sección se presentará una refactorización guiada, paso a paso, que permita al estudiante comprender cómo transformar una clase Dios en un conjunto de clases bien diseñadas, aplicando los criterios estudiados en las secciones anteriores.

Punto de partida: una clase Dios funcional pero mal diseñada

Retomemos un ejemplo típico, similar a los analizados anteriormente. Supóngase un sistema de gestión académica simplificado con la siguiente clase central:

```
class SistemaAcademico {

    public void registrarEstudiante() {
        validarDatos();
        guardarEstudiante();
        enviarCorreo();
    }

    public void validarDatos() {
        // validaciones
    }

    public void guardarEstudiante() {
        // persistencia
    }
```

```
    public void enviarCorreo() {
        // notificación
    }

    public void generarReporte() {
        // reporte académico
    }
}
```


Esta clase funciona, pero presenta todos los síntomas de una clase Dios: concentra validaciones, persistencia, comunicación y generación de reportes en un único lugar.

El objetivo de la refactorización será dividir estas responsabilidades sin romper el sistema.

- Paso 1: identificar responsabilidades explícitas

Antes de escribir nuevo código, es fundamental analizar lo que la clase hace realmente, no solo cómo se llama. En este caso, se pueden identificar claramente las siguientes responsabilidades:

- Representar el proceso de registro de estudiantes.
- Validar información.
- Guardar datos.
- Enviar notificaciones.
- Generar reportes.
- Cada una de estas responsabilidades puede y debe ser gestionada por una clase distinta.

Este paso es conceptual y no implica todavía cambios en el código. Su finalidad es ordenar mentalmente el problema, una habilidad clave en la Programación Orientada a Objetos.

- Paso 2: separar el modelo del proceso

El primer error común en clases Dios es que no existe una clase que represente claramente la entidad principal del dominio. En este caso, el estudiante no está modelado explícitamente.

Se comienza creando una clase que represente únicamente al estudiante:

```
class Estudiante {
    private String nombre;
    private String correo;

    public Estudiante(String nombre, String correo) {
        this.nombre = nombre;
        this.correo = correo;
    }

    public String getCorreo() {
        return correo;
    }
}
```

Esta clase tiene una responsabilidad clara: representar al estudiante y sus datos. No valida, no guarda, no envía correos.

- Paso 3: extraer validaciones

Las validaciones constituyen una responsabilidad independiente y suelen ser una de las primeras candidatas para extraerse de una clase Dios.

```
class ValidadorEstudiante {


    public boolean validar(Estudiante estudiante) {
        // reglas de validación
        return true;
    }

}
```

Con esta separación, cualquier cambio en las reglas de validación se realiza en un único lugar, sin afectar la lógica del sistema académico.

- Paso 4: separar la persistencia

Guardar información es otra responsabilidad claramente diferenciada. Esta tarea no debe recaer ni en el modelo ni en el gestor del proceso.

```
class RepositorioEstudiante {


    public void guardar(Estudiante estudiante) {
        // guardar en archivo o base de datos
    }

}
```

Esta clase puede cambiar internamente (archivo, base de datos, servicio externo) sin afectar al resto del sistema.

- Paso 5: aislar la comunicación

El envío de notificaciones es una responsabilidad transversal que no pertenece al núcleo del modelo.

```
class ServicioCorreo {

    public void enviarCorreoBienvenida(Estudiante estudiante) {
        // envío de correo
    }
}
```

Gracias a esta separación, es posible agregar nuevos canales de comunicación sin modificar otras clases.

- Paso 6: crear un coordinador del proceso

Una vez separadas las responsabilidades, es necesario un componente que coordine el flujo, sin asumir tareas que no le corresponden. Esta clase actúa como orquestadora del proceso.

```
class GestorAcademico {

    private ValidadorEstudiante validador;
    private RepositorioEstudiante repositorio;
    private ServicioCorreo servicioCorreo;

    public GestorAcademico(ValidadorEstudiante validador,
                           RepositorioEstudiante repositorio,
                           ServicioCorreo servicioCorreo) {
```

```
        this.validador = validador;
        this.repositorio = repositorio;
        this.servicioCorreo = servicioCorreo;
    }

    public void registrarEstudiante(Estudiante estudiante) {
        if (validador.validar(estudiante)) {
            repositorio.guardar(estudiante);
            servicioCorreo.enviarCorreoBienvenida(estudiante);
        }
    }
}
```

Esta clase no valida directamente, no guarda directamente, no envía correos directamente. Su única responsabilidad es coordinar el proceso de registro.

Comparación antes y después de la refactorización

Tabla 2.1. Comparación antes y después de la refactorización.

Aspecto	Clase Dios	Diseño refactorizado
Responsabilidades	Múltiples	Una por clase
Tamaño de clases	Grande	Pequeñas y claras
Mantenibilidad	Baja	Alta
Reutilización	Difícil	Sencilla
Claridad conceptual	Baja	Alta

Esta comparación (Tabla 2.1) evidencia que la refactorización no agrega complejidad funcional, sino que reduce la complejidad estructural.

Beneficios pedagógicos y técnicos

Desde el punto de vista del aprendizaje, esta refactorización enseña al estudiante a:

- Pensar el sistema como un conjunto de responsabilidades.
- Diseñar clases con propósito claro.
- Evitar soluciones monolíticas.
- Prepararse para sistemas más grandes y reales.

Desde el punto de vista técnico, el sistema resultante es más flexible, extensible y robusto.

Ejercicio guiado de aplicación: Refactorización paso a paso en un caso real

Contexto del caso: Sistema de pedidos en una cafetería universitaria

En una cafetería universitaria se desea implementar un sistema sencillo para registrar pedidos. El sistema debe permitir:

1. Registrar un pedido con:
    - nombre del cliente,
    - lista de productos (ej.: café, empanada, jugo),
    - total a pagar.
2. Validar que:
    - el cliente tenga nombre,
    - exista al menos un producto,
    - el total sea mayor que cero.
3. Guardar el pedido (por ahora, en memoria o simulando un archivo).

4. Imprimir un comprobante (por consola).
5. Enviar una notificación simple al cliente (por consola), simulando un SMS o WhatsApp.

Importante: este caso es realista porque mezcla tareas típicas que los estudiantes suelen juntar en una sola clase.

- **Paso 0: Diseño inicial incorrecto (clase monolítica)**

Un estudiante normalmente intentaría resolverlo así:

```
import java.util.ArrayList;
import java.util.List;


class SistemaCafeteria {


    private List<String> pedidosGuardados = new Array-
List<>();


    public void registrarPedido(String cliente, List<String>
productos, double total) {


        // 1) Validar datos
        if (cliente == null || cliente.trim().isEmpty()) {
            System.out.println("Error: Cliente inválido". );
            return;
        }
        if (productos == null || productos.isEmpty()) {
```

```
        System.out.println("Error: Debe existir al menos un
producto". );
            return;
        }
        if (total <= 0) {
            System.out.println("Error: Total inválido". );
            return;
        }

        // 2) Guardar pedido (simulado)
       String pedidoTexto = "Cliente: " + cliente + ", Productos:
" + productos + ", Total: $" + total;
        pedidosGuardados.add(pedidoTexto);

        // 3) Imprimir comprobante
        System.out.println("=== COMPROBANTE ===");
        System.out.println(pedidoTexto);
        System.out.println("==================");

        // 4) Enviar notificación (simulada)
        System.out.println("Notificación enviada a " + cliente +
": Pedido registrado correctamente". );
    }
}
```

¿Qué está mal aquí?

Esta clase hace demasiadas cosas:

- valida,
- construye el pedido,
- guarda,
- imprime comprobante,
- envía notificación.

Es funcional, pero es un diseño que no se puede mantener fácilmente.

Refactorización guiada

A continuación, se muestra cómo refactorizar correctamente, conservando el comportamiento pero mejorando el diseño.

- **Paso 1: Crear el modelo del dominio (la clase Pedido)**

Primero, identificamos el concepto central del problema: Pedido.

Regla: una clase del dominio debe representar datos y comportamiento propio del dominio, no persistencia ni impresión ni notificaciones.

```
import java.util.List;


class Pedido {
    private String cliente;
    private List<String> productos;
    private double total;
```

```
    public Pedido(String cliente, List<String> productos, dou-
ble total) {
        this.cliente = cliente;
        this.productos = productos;
        this.total = total;
    }

    public String getCliente() {
        return cliente;
    }

    public List<String> getProductos() {
        return productos;
    }

    public double getTotal() {
        return total;
    }
}
```

Ahora se tiene una entidad clara y reutilizable.

- **Paso 2: Extraer las validaciones (ValidadorPedido)**

Luego, se separa la responsabilidad de validar. Esto permite que las reglas de validación crezcan sin ensuciar el flujo principal.

```
class ValidadorPedido {

    public boolean validar(Pedido pedido) {
        if (pedido == null) return false;

          if (pedido.getCliente() == null || pedido.getCliente().
trim().isEmpty()) {
            System.out.println("Error: Cliente inválido". );
            return false;
        }

         if (pedido.getProductos() == null || pedido.getProduc-
tos().isEmpty()) {
             System.out.println("Error: Debe existir al menos un
producto". );
            return false;
        }

        if (pedido.getTotal() <= 0) {
            System.out.println("Error: Total inválido". );
            return false;
        }

        return true;
    }
}
```

- **Paso 3: Separar la persistencia (RepositorioPedido)**

Ahora se separa el almacenamiento. Aquí puede ser "memoria", "archivo", "base de datos", etc. La clave es que el resto del sistema no debe saber cómo se guarda.

```
import java.util.ArrayList;
import java.util.List;

class RepositorioPedido {

    private List<Pedido> pedidos = new ArrayList<>();

    public void guardar(Pedido pedido) {
        pedidos.add(pedido);
    }

    public List<Pedido> listarPedidos() {
        return pedidos;
    }
}
```

Persistencia desacoplada del resto.

- **Paso 4: Separar la impresión del comprobante (ServicioComprobante)**

La impresión del comprobante no debe estar en la lógica del sistema. Se crea un servicio específico.

```
class ServicioComprobante {

    public void imprimir(Pedido pedido) {
        System.out.println("=== COMPROBANTE ===");
        System.out.println("Cliente: " + pedido.getCliente());
        System.out.println("Productos: " + pedido.getProductos());
        System.out.println("Total: $" + pedido.getTotal());
        System.out.println("==================");
    }
}
```

Ahora el comprobante puede cambiar de formato sin tocar el flujo principal.

- **Paso 5: Separar la notificación (ServicioNotificacion)**

Se crea un servicio para notificaciones.

```
class ServicioNotificacion {

    public void enviarConfirmacion(Pedido pedido) {
        System.out.println("Notificación enviada a " + pedido.getCliente() +
                ": Pedido registrado correctamente. Total $" + pedido.getTotal());
    }
}
```

Se puede cambiar por SMS, email, WhatsApp sin afectar la lógica de registro.

- **Paso 6: Crear el coordinador del flujo (GestorPedidos)**

Este es el paso más importante: crear una clase que coordine el proceso, pero no haga tareas que no le corresponden.

```
class GestorPedidos {

    private ValidadorPedido validador;
    private RepositorioPedido repositorio;
    private ServicioComprobante comprobante;
    private ServicioNotificacion notificacion;

    public GestorPedidos(ValidadorPedido validador,
                         RepositorioPedido repositorio,
                         ServicioComprobante comprobante,
                         ServicioNotificacion notificacion) {
        this.validador = validador;
        this.repositorio = repositorio;
        this.comprobante = comprobante;
        this.notificacion = notificacion;
    }

    public void registrarPedido(Pedido pedido) {
```

```
            if (validador.validar(pedido)) {
                repositorio.guardar(pedido);
                comprobante.imprimir(pedido);
                notificacion.enviarConfirmacion(pedido);
            }
        }
    }
```

Esta clase tiene una responsabilidad clara: coordinar el registro de pedidos.

- **Paso 7: Programa principal de prueba (Main) para que el estudiante lo ejecute**

Este main permite al estudiante probar y verificar que todo funciona.

```
import java.util.Arrays;


public class MainCafeteria {


    public static void main(String[] args) {


        // Crear componentes
        ValidadorPedido validador = new ValidadorPedido();
        RepositorioPedido repositorio = new RepositorioPedi-
do();
        ServicioComprobante comprobante = new Servicio-
Comprobante();
        ServicioNotificacion notificacion = new ServicioNotifi-
```

```
cacion();

        // Crear gestor
        GestorPedidos gestor = new GestorPedidos(validador,
repositorio, comprobante, notificacion);

        // Crear pedido realista
        Pedido pedido = new Pedido(
            "Ana Torres",
             Arrays.asList("Café Americano", "Empanada de
queso", "Jugo de naranja"),
            4.75
        );

        // Registrar pedido
        gestor.registrarPedido(pedido);

        // Verificar persistencia (opcional)
        System.out.println("Pedidos guardados: " + repositorio.
listarPedidos().size());
    }
}
```


Con este ejercicio, el estudiante puede programar y sistematizar que:

1. Un sistema puede funcionar estando mal diseñado.

2. Refactorizar significa reorganizar responsabilidades.
3. Una clase del dominio (Pedido) no debe:

- validar,
- guardar,
- imprimir,
- notificar.

4. El flujo de trabajo debe coordinarse desde una clase que no haga todo, sino que delegue.

A través de estas refactorizaciones guiadas se ha demostrado que un mal diseño puede corregirse sin reescribir el sistema, simplemente reorganizando responsabilidades y aplicando criterios de cohesión y separación de tareas. Este proceso permite transformar clases grandes y confusas en estructuras limpias y comprensibles.

Sin embargo, refactorizar correctamente no solo implica dividir clases, sino también evaluar qué tan bien están relacionadas internamente y qué tan dependientes son unas de otras. Para profundizar en estos aspectos, en la siguiente sección se analizarán con mayor detalle los conceptos de cohesión y acoplamiento, fundamentales para evaluar la calidad de un diseño orientado a objetos.

2.5. Cohesión y acoplamiento en el diseño orientado a objetos

Después de analizar cómo distribuir responsabilidades y cómo refactorizar clases mal diseñadas, es necesario introducir dos conceptos fundamentales que permiten evaluar la calidad de un diseño orientado a objetos: la cohesión y el acoplamiento. Estos conceptos actúan como criterios prácticos para decidir si una solución está bien estructurada o si, por el contrario, presenta

problemas que dificultarán su mantenimiento y evolución.

Mientras que la responsabilidad única se centra en qué debe hacer una clase, la cohesión y el acoplamiento permiten analizar cómo de bien está organizada internamente y qué tan dependiente es de otras clases.

Cohesión: ¿qué tan unida está una clase?

La cohesión mide qué tan relacionados están los atributos y métodos dentro de una clase. Una clase con alta cohesión contiene elementos que trabajan juntos para cumplir una única responsabilidad claramente definida. Por el contrario, una clase con baja cohesión agrupa funcionalidades que no guardan relación directa entre sí.

En términos prácticos, una clase cohesionada responde con claridad a la pregunta ¿Todos los métodos de esta clase existen para apoyar el mismo propósito?

Ejemplo de alta cohesión (modelo académico)

```
class Curso {
    private String nombre;
    private int creditos;
    private int cupoMaximo;
    private int inscritos;

    public Curso(String nombre, int creditos, int cupoMaximo)
{
        this.nombre = nombre;
        this.creditos = creditos;
        this.cupoMaximo = cupoMaximo;
```

```
        this.inscritos = 0;
    }

    public boolean hayCupoDisponible() {
        return inscritos < cupoMaximo;
    }

    public void inscribirEstudiante() {
        if (hayCupoDisponible()) {
            inscritos++;
        }
    }

    public int getInscritos() {
        return inscritos;
    }
}
```

En este ejemplo:

- Todos los atributos están relacionados con la gestión de un curso.
- Todos los métodos interactúan directamente con esos atributos.
- No existen métodos ajenos al concepto Curso.

Esta clase presenta alta cohesión, ya que cada elemento contribuye a una misma responsabilidad.

Ejemplo de baja cohesión (mezcla de responsabilidades)

```
class Curso {
    private String nombre;
    private int creditos;
    private int cupoMaximo;
    private int inscritos;

    public void inscribirEstudiante() { }

    public void generarReportePDF() { }

    public void enviarCorreoAviso() { }

    public void guardarEnArchivo() { }
}
```

Aquí la clase mezcla:

- Lógica académica.
- Generación de reportes.
- Comunicación.
- Persistencia.

Aunque el nombre de la clase es Curso, su comportamiento ya no representa únicamente ese concepto, lo que reduce significativamente su cohesión.

Consecuencias de una baja cohesión

Una clase con baja cohesión suele presentar los siguientes problemas:

- Dificultad para entender su propósito.
- Mayor probabilidad de errores al modificarla.
- Clases grandes y difíciles de probar.
- Incremento del acoplamiento con otras partes del sistema.

Por esta razón, buscar alta cohesión es un objetivo central en el diseño orientado a objetos.

Acoplamiento: dependencia entre clases

El acoplamiento mide el grado de dependencia entre clases. Un sistema con alto acoplamiento tiene clases fuertemente dependientes entre sí, de modo que un cambio en una clase afecta directamente a muchas otras. En contraste, un sistema con bajo acoplamiento mantiene dependencias mínimas y controladas.

Desde una perspectiva práctica, cuanto menos conozca una clase sobre las demás, mejor diseñado estará el sistema.

Ejemplo de alto acoplamiento (sistema de inventario)

```
class Inventario {


    public void actualizarStock() {
        BaseDatosMySQL db = new BaseDatosMySQL();
        db.conectar();
        db.ejecutarConsulta("UPDATE stock..". );
    }
}
```

En este ejemplo:

- La clase Inventario depende directamente de una implementación concreta (BaseDatosMySQL).
- Si cambia la base de datos, la clase debe modificarse.
- No existe flexibilidad ni aislamiento.

Este diseño presenta alto acoplamiento, ya que la clase conoce detalles que no le corresponden.

Ejemplo de bajo acoplamiento (mismo problema, mejor diseño)

```
interface RepositorioInventario {
    void actualizarStock();
}
class RepositorioInventarioMySQL implements RepositorioInventario {

    public void actualizarStock() {
        // implementación concreta
    }
}
class Inventario {

    private RepositorioInventario repositorio;

    public Inventario(RepositorioInventario repositorio) {
        this.repositorio = repositorio;
```

```
    }

    public void actualizarStock() {
        repositorio.actualizarStock();
    }
}
```


Aquí:

- Inventario depende de una abstracción, no de una implementación concreta.
- Se puede cambiar la base de datos sin modificar la clase.
- El diseño es más flexible y mantenible.

Este es un ejemplo claro de bajo acoplamiento.

Relación entre cohesión y acoplamiento

Estos dos conceptos están estrechamente relacionados:

- Alta cohesión suele conducir a bajo acoplamiento.
- Baja cohesión suele provocar alto acoplamiento.

Cuando una clase tiene una responsabilidad clara, necesita menos información de otras clases. Por el contrario, cuando una clase hace demasiadas cosas, termina dependiendo de muchas otras.

En esta sección se ha demostrado que un buen diseño orientado a objetos no depende únicamente de que el código funcione, sino de cómo están organizadas internamente las clases y cómo se relacionan entre sí. La cohesión y el acoplamiento permiten evaluar objetivamente la calidad estructural de una solución.

Comprender y aplicar estos conceptos prepara al estudiante para abordar sistemas más complejos y para tomar decisiones de diseño más acertadas. En la siguiente sección del capítulo se integrarán estos criterios en un caso final de análisis, donde se evaluará un diseño completo utilizando responsabilidad única, refactorización, cohesión y acoplamiento como herramientas de evaluación.

2.6. Caso integrador final del Capítulo 2

"Sistema de Turnos para Clínica Dental": diseño, detección de problemas y refactorización justificada.

En este caso integrador se aplicarán, de manera conjunta, los conceptos trabajados a lo largo del capítulo:

- Buena clase (propósito claro)
- Responsabilidad única
- Evitar clases monolíticas
- Refactorización guiada
- Evaluación mediante cohesión y acoplamiento

El objetivo no es solo ver código, sino aprender a pensar el diseño, justificar decisiones y comprender por qué una estructura es mejor que otra.

Contexto de la vida real

Una clínica dental necesita un sistema sencillo para gestionar turnos (citas). Se desea:

1. Registrar pacientes.
2. Agendar turnos con fecha y hora.
3. Validar reglas:

- El paciente debe tener cédula y nombre.

- No se puede agendar un turno en el pasado.
- Un odontólogo no puede tener dos turnos en la misma hora.

4. Calcular el costo aproximado del turno según el servicio:

- Limpieza: $20.
- Extracción: $35.
- Ortodoncia: $50.

5. Guardar el turno (por ahora en memoria, simulando persistencia).
6. Generar una confirmación del turno (impresión por consola).
7. Enviar notificación de confirmación (simulada por consola).

Este caso es realista porque, en la práctica, muchos estudiantes intentarían resolver todo dentro de una sola clase.

Diseño inicial típico (mal diseño): clase monolítica

Un diseño común (y problemático) es crear una clase que "lo controla todo":

```
import java.util.*;


class SistemaClinicaDental {


  private List<String> turnosGuardados = new ArrayList<>();


    public void agendarTurno(String cedulaPaciente, String
nombrePaciente,
```

```
                String odontologo, String servicio,
                Date fechaHora) {

    // 1) Validaciones
        if (cedulaPaciente == null || cedulaPaciente.trim().
isEmpty()) {
        System.out.println("Error: cédula inválida". );
        return;
    }
      if (nombrePaciente == null || nombrePaciente.trim().
isEmpty()) {
        System.out.println("Error: nombre inválido". );
        return;
    }
    if (fechaHora.before(new Date())) {
         System.out.println("Error: no se puede agendar en
el pasado". );
        return;
    }

    // 2) Regla de choque de turnos (simplificada)
    for (String t : turnosGuardados) {
        if (t.contains("Odontologo:" + odontologo) && t.con-
tains("Fecha:" + fechaHora.toString())) {
            System.out.println("Error: el odontólogo ya tiene
un turno en esa fecha/hora". );
```

```
            return;
        }
    }

    // 3) Calcular costo
    double costo = 0;
    if (servicio.equalsIgnoreCase("Limpieza")) costo = 20;
    else if (servicio.equalsIgnoreCase("Extraccion")) costo
= 35;
    else if (servicio.equalsIgnoreCase("Ortodoncia")) costo
= 50;

    // 4) Guardar turno (simulado)
     String turno = "Paciente:" + nombrePaciente + " (" +
cedulaPaciente + "), " +
                "Odontologo:" + odontologo + ", Servicio:" +
servicio +
                ", Fecha:" + fechaHora + ", Costo:$" + costo;
    turnosGuardados.add(turno);

    // 5) Confirmación (impresión)
        System.out.println("=== CONFIRMACIÓN TURNO
===");
    System.out.println(turno);
   System.out.println("==========================");

    // 6) Notificación (simulada)
```

```
        System.out.println("Notificación enviada a " + nombre-
    Paciente + ": Turno confirmado". );
      }
    }
```

¿Qué problemas tiene este diseño?

Esta clase está haciendo demasiadas cosas:

- valida datos del paciente,
- controla reglas del negocio (choque de turnos),
- calcula costos,
- gestiona almacenamiento,
- imprime confirmaciones,
- envía notificaciones.

Eso significa:

- Baja cohesión: métodos y responsabilidades mezcladas.
- Alto acoplamiento: la clase depende de detalles (impresión, notificación, reglas, almacenamiento).
- Difícil de mantener: si cambia el costo del servicio o el medio de notificación, hay que tocar esta clase.

Esto es exactamente el patrón de clase monolítica (clase Dios) visto en 2.4.

Refactorización guiada y justificada (paso a paso)

- **Paso 1: Crear las clases del dominio (modelos)**

Primero se modela el dominio.

En una clínica dental, existen conceptos claros: Paciente, Turno, y el tipo de servicio.

```
Clase Paciente
class Paciente {
    private String cedula;
    private String nombre;

    public Paciente(String cedula, String nombre) {
        this.cedula = cedula;
        this.nombre = nombre;
    }

    public String getCedula() { return cedula; }
    public String getNombre() { return nombre; }
}
```

Justificación: El paciente solo representa información, no valida, no guarda, no notifica.

Enum ServicioDental

Usar enum hace el sistema más claro y evita errores típicos por cadenas mal escritas.

```
enum ServicioDental {
    LIMPIEZA(20),
    EXTRACCION(35),
    ORTODONCIA(50);

    private final double costo;
```

```
    ServicioDental(double costo) {
        this.costo = costo;
    }

    public double getCosto() {
        return costo;
    }
}
```

Justificación:

- Evita escribir “Limpieza” o “limpiesa” y que el sistema falle.
- Permite mantener costos centralizados y consistentes.

Clase Turno

```
import java.util.Date;

class Turno {
    private Paciente paciente;
    private String odontologo;
    private ServicioDental servicio;
    private Date fechaHora;

    public Turno(Paciente paciente, String odontologo, ServicioDental servicio, Date fechaHora) {
```

```
        this.paciente = paciente;
        this.odontologo = odontologo;
        this.servicio = servicio;
        this.fechaHora = fechaHora;
    }

    public Paciente getPaciente() { return paciente; }
    public String getOdontologo() { return odontologo; }
    public ServicioDental getServicio() { return servicio; }
    public Date getFechaHora() { return fechaHora; }

    public double calcularCosto() {
        return servicio.getCosto();
    }
}
```

Justificación:

- El costo está asociado al turno (depende del servicio).
- Esto mejora cohesión: "Turno" agrupa datos del turno y comportamiento coherente.

- **Paso 2: Extraer validaciones (responsabilidad única)**

```
import java.util.Date;

class ValidadorTurno {

    public boolean validarTurno(Turno turno) {
```

```
if (turno == null) return false;

if (turno.getPaciente() == null) {
    System.out.println("Error: paciente no definido". );
    return false;
}

if (turno.getPaciente().getCedula() == null || turno.get-
Paciente().getCedula().trim().isEmpty()) {
    System.out.println("Error: cédula inválida". );
    return false;
}

if (turno.getPaciente().getNombre() == null || turno.
getPaciente().getNombre().trim().isEmpty()) {
    System.out.println("Error: nombre inválido". );
    return false;
}

Date ahora = new Date();
if (turno.getFechaHora() == null || turno.getFechaHo-
ra().before(ahora)) {
    System.out.println("Error: no se puede agendar un
turno en el pasado". );
    return false;
```

```
        }

        if (turno.getOdontologo() == null || turno.getOdontolo-
    go().trim().isEmpty()) {
            System.out.println("Error: odontólogo inválido". );
            return false;
        }

        return true;
    }
}
```

Justificación:

1. Las validaciones cambian con frecuencia (nuevas reglas).
2. Separarlas evita que el gestor principal se vuelva monolítico.

- **Paso 3: Separar almacenamiento (repositorio en memoria)**

```
import java.util.ArrayList;
import java.util.List;

class RepositorioTurnos {

    private List<Turno> turnos = new ArrayList<>();

    public void guardar(Turno turno) {
        turnos.add(turno);
```

```
    }

    public List<Turno> listar() {
        return turnos;
    }
}
```


Justificación:

1. Hoy es memoria. Mañana puede ser archivo o base de datos.
2. El resto del sistema no debería cambiar por eso.

- **Paso 4: Regla de choque de turnos (servicio de negocio)**

Esta regla pertenece a la lógica del dominio, pero no al repositorio ni al validador.

```
class ServicioDisponibilidad {

    private RepositorioTurnos repositorio;

    public ServicioDisponibilidad(RepositorioTurnos repositorio) {
        this.repositorio = repositorio;
    }

    public boolean hayChoque(Turno nuevoTurno) {
        for (Turno t : repositorio.listar()) {
            boolean mismoOdontologo = t.getOdontologo().equalsIgnoreCase(nuevoTurno.getOdontologo());
```

```
        boolean mismaFechaHora = t.getFechaHora().equal-
s(nuevoTurno.getFechaHora());


            if (mismoOdontologo && mismaFechaHora) {
                return true;
            }
        }
        return false;
    }
}
```

Justificación:

1. La regla de disponibilidad puede cambiar (ej.: permitir 15 min de diferencia).
2. Mantenerla separada hace el sistema adaptable.

- **Paso 5: Confirmación y notificación (servicios de salida)**

```
class ServicioConfirmacion {


    public void imprimirConfirmacion(Turno turno) {
        System.out.println("=== CONFIRMACIÓN TURNO
===");
        System.out.println("Paciente: " + turno.getPaciente().
getNombre() + " (" + turno.getPaciente().getCedula() + ")");
        System.out.println("Odontólogo: " + turno.getOdonto-
logo());
        System.out.println("Servicio: " + turno.getServicio());
```

```
        System.out.println("Fecha y hora: " + turno.getFe-
chaHora());
        System.out.println("Costo: $" + turno.calcularCosto());
        System.out.println("========================");
    }
}
class ServicioNotificacion {

    public void enviarNotificacion(Turno turno) {
        System.out.println("Notificación enviada a " + turno.
getPaciente().getNombre() +
            ": Turno confirmado para " + turno.getFechaHora()
+
            " (" + turno.getServicio() + ")". );
    }
}
```

Justificación: Imprimir y notificar son salidas del sistema, no deberían mezclarse con validaciones o persistencia.

Coordinador final (diseño limpio)

Ahora se crea una clase que coordina el proceso de agendar, sin hacer "de todo".

```
class GestorTurnos {

    private ValidadorTurno validador;
    private RepositorioTurnos repositorio;
```

```
private ServicioDisponibilidad disponibilidad;
private ServicioConfirmacion confirmacion;
private ServicioNotificacion notificacion;

public GestorTurnos(ValidadorTurno validador,
                    RepositorioTurnos repositorio,
                    ServicioDisponibilidad disponibilidad,
                    ServicioConfirmacion confirmacion,
                    ServicioNotificacion notificacion) {
    this.validador = validador;
    this.repositorio = repositorio;
    this.disponibilidad = disponibilidad;
    this.confirmacion = confirmacion;
    this.notificacion = notificacion;
}

public void agendarTurno(Turno turno) {

    // 1) validar datos
    if (!validador.validarTurno(turno)) return;

    // 2) verificar disponibilidad
    if (disponibilidad.hayChoque(turno)) {
        System.out.println("Error: el odontólogo ya tiene un
```

```
        turno en esa fecha/hora". );
            return;
        }

        // 3) guardar
        repositorio.guardar(turno);

        // 4) confirmar y notificar
        confirmacion.imprimirConfirmacion(turno);
        notificacion.enviarNotificacion(turno);
    }
}
```

Justificación:

1. Esta clase tiene alta cohesión porque su propósito es agendar turnos.
2. Tiene bajo acoplamiento porque depende de servicios específicos, no mezcla responsabilidades.

Programa principal de prueba (para que el estudiante lo ejecute)

```
import java.util.Date;

public class MainClinicaDental {

    public static void main(String[] args) {
```

```
// Componentes
RepositorioTurnos repositorio = new RepositorioTurnos();
ValidadorTurno validador = new ValidadorTurno();
ServicioDisponibilidad disponibilidad = new ServicioDisponibilidad(repositorio);
ServicioConfirmacion confirmacion = new ServicioConfirmacion();
ServicioNotificacion notificacion = new ServicioNotificacion();

// Gestor
GestorTurnos gestor = new GestorTurnos(validador, repositorio, disponibilidad, confirmacion, notificacion);

// Datos de prueba (fecha futura)
Date fechaFutura = new Date(System.currentTimeMillis() + 60 * 60 * 1000); // +1 hora

Paciente p1 = new Paciente("1723456789", "María Castillo");
Turno t1 = new Turno(p1, "Dr. Pérez", ServicioDental.LIMPIEZA, fechaFutura);

gestor.agendarTurno(t1);

// Intento de choque: mismo odontólogo y misma fecha/
```

```
hora

        Paciente p2 = new Paciente("0912345678", "Carlos
Mena");

        Turno t2 = new Turno(p2, "Dr. Pérez", ServicioDental.
EXTRACCION, fechaFutura);


      gestor.agendarTurno(t2);
    }
}
```


Evaluación del diseño con cohesión y acoplamiento (explicación para el estudiante)

Cohesión

- Paciente: cohesión alta (solo representa al paciente).
- Turno: cohesión alta (datos + cálculo coherente del costo).
- ValidadorTurno: cohesión alta (solo valida).
- RepositorioTurnos: cohesión alta (solo almacena).
- GestorTurnos: cohesión alta (coordina el proceso de agendar).

Acoplamiento

- Ninguna clase depende directamente de detalles de otra fuera de lo necesario.
- El repositorio podría cambiar sin alterar el validador o la notificación.
- La confirmación podría imprimirse distinto sin tocar el agendamiento.

Conclusión: diseño limpio, escalable en mantenimiento y fácil de extender.

Este caso integrador demuestra que una solución orientada a objetos no se evalúa únicamente por si compila o funciona, sino por su estructura interna. Una clase monolítica puede funcionar hoy, pero se vuelve difícil de modificar mañana. Al separar responsabilidades, aumentar cohesión y reducir acoplamiento, se obtiene un diseño más profesional, más fácil de probar y más fácil de mantener.

En este capítulo se abordó uno de los aspectos más importantes de la Programación Orientada a Objetos: el diseño correcto de clases y la asignación adecuada de responsabilidades. Más allá de la sintaxis de un lenguaje de programación, se puso énfasis en la forma de pensar el software y en cómo las decisiones de diseño influyen directamente en la claridad, mantenibilidad y evolución de un sistema.

Se inició analizando qué se entiende por una buena clase, destacando la necesidad de que cada clase represente un concepto claro del dominio del problema, con coherencia entre su nombre, sus atributos y sus métodos. Se enfatizó que una clase bien diseñada debe ser comprensible por sí misma y tener un propósito claramente definido.

Posteriormente, se introdujo el criterio de responsabilidad única, explicando que una clase debería tener una sola razón principal para cambiar. A través de ejemplos prácticos, se evidenció cómo la acumulación de responsabilidades conduce a diseños frágiles y difíciles de mantener, aun cuando el sistema funcione correctamente.

A continuación, se analizó uno de los errores de diseño más comunes: las clases monolíticas o clases Dios. Se mostró cómo estas clases concentran múltiples

tareas que no les corresponden y cómo su uso afecta negativamente la cohesión, incrementa el acoplamiento y dificulta la evolución del sistema.

El capítulo avanzó luego hacia la refactorización guiada, demostrando paso a paso cómo transformar una clase mal diseñada en un conjunto de clases más pequeñas, coherentes y con responsabilidades bien delimitadas, sin modificar el comportamiento funcional del sistema. Este proceso permitió comprender que mejorar el diseño no implica reescribir el programa, sino reorganizarlo de forma más inteligente.

Seguidamente, se desarrollaron los conceptos de cohesión y acoplamiento como herramientas de evaluación del diseño orientado a objetos. Se explicó que una alta cohesión y un bajo acoplamiento son indicadores clave de calidad estructural y que estos criterios permiten anticipar problemas de mantenimiento antes de que se manifiesten en el código.

Finalmente, el caso integrador del sistema de turnos para una clínica dental permitió aplicar de manera conjunta todos los conceptos estudiados. A través de un ejemplo de la vida real, se justificaron decisiones de diseño, se evaluaron alternativas y se consolidó una forma de pensar orientada a la construcción de software limpio, modular y comprensible.

En conjunto, este capítulo proporciona una base sólida para que el estudiante desarrolle una mentalidad de diseño orientada a objetos, capaz de identificar problemas estructurales, proponer mejoras y construir soluciones más profesionales. Estos aprendizajes serán fundamentales para los capítulos siguientes, donde se profundizará en el uso adecuado de herencia, polimorfismo e interfaces dentro de sistemas más complejos.

03

Diseño extensible en Programación Orientada a Objetos: cómo crecer sin romper el sistema

3.1. El error más común al extender sistemas: la explosión de condicionales

En el Capítulo 1 se establecieron los conceptos fundamentales de la Programación Orientada a Objetos, incluyendo UML y las relaciones entre clases (asociación, agregación, composición y generalización), además de un repaso de herencia, polimorfismo y abstracción con ejemplos y casos de uso.

En el Capítulo 2 se profundizó en un tema decisivo: el diseño correcto de clases y responsabilidades. Se trabajó cómo identificar clases bien cohesionadas, evitar clases monolíticas (clases Dios), refactorizar diseños deficientes y evaluar calidad estructural mediante cohesión y acoplamiento.

Con esas bases, el estudiante ya puede crear sistemas ordenados. Sin embargo, los proyectos reales siempre enfrentan un fenómeno inevitable: Los requisitos cambian y el sistema debe extenderse.

El mayor problema en proyectos académicos y profesionales no es escribir código que funcione hoy, sino construir software que pueda evolucionar sin convertirse en un caos. Este capítulo aborda precisamente esa habilidad: diseñar sistemas extensibles, aplicando los conceptos anteriores para que el crecimiento sea controlado.

En la práctica, cuando aparece un nuevo tipo o una nueva regla, muchos estudiantes recurren a:

- aumentar if/else y switch,
- duplicar métodos,
- agrandar clases hasta volverlas monolíticas,
- mezclar lógica de negocio con persistencia y presentación.

Esto provoca exactamente lo que se evitó en el Capítulo 2: baja cohesión, alto acoplamiento y mal mantenimiento. Por ello, este capítulo enseña a manejar la variación con diseño limpio:

- agregar clases sin modificar las existentes,
- encapsular lo que cambia,
- mantener estabilidad en el núcleo del sistema.

Para comenzar, analizaremos el error más común cuando el sistema crece: la "explosión de condicionales". Ese problema se estudiará con un ejemplo realista y se mostrará cómo corregirlo con un diseño extensible.

Cuando un sistema comienza a crecer, no lo hace de manera abstracta, sino a través de variaciones concretas: nuevos tipos, nuevas reglas, nuevos escenarios de uso. Este crecimiento es natural y esperable en cualquier proyecto real, académico o profesional. Sin embargo, la forma en que se responde a estas variaciones marca la diferencia entre un sistema que evoluciona de forma ordenada y uno que se degrada progresivamente.

Uno de los errores más frecuentes y más peligrosos al enfrentar este crecimiento es recurrir a una solución rápida basada en condicionales acumulativos (if, else if, switch). A primera vista, esta estrategia parece lógica y sencilla, especialmente para estudiantes en etapas tempranas de aprendizaje. El problema no es que los condicionales sean incorrectos en sí mismos, sino que no escalan bien cuando el sistema evoluciona.

La ilusión de control inicial

En versiones tempranas del sistema, suele existir una sensación de control:

- "Solo hay dos o tres casos".

- “Es más rápido hacerlo aquí mismo”.
- “Luego lo mejoramos”.

En este punto, el código suele ser corto y aparentemente claro. El error comienza cuando estas decisiones temporales se convierten en estructurales. Cada nuevo requerimiento se resuelve añadiendo otro bloque condicional, sin replantear el diseño general.

Con el tiempo, la clase o el método central se transforma en un punto único donde convergen todas las decisiones del sistema.

¿Qué es realmente la “explosión de condicionales”?

La explosión de condicionales ocurre cuando:

- Un método concentra muchas decisiones de negocio.
- Cada nueva variante implica agregar una condición adicional.
- El flujo del programa depende de cadenas largas de decisiones.
- El comportamiento del sistema se define más por casos que por objetos.

Este fenómeno no se limita a un solo lugar del código. Frecuentemente aparece en:

- Cálculos de precios.
- Procesos de validación.
- Flujos de pago.
- Generación de reportes.
- Manejo de estados o tipos.

En lugar de modelar el problema mediante objetos con comportamiento propio, el sistema comienza a comportarse como una tabla de decisiones codificada manualmente.

El problema no es el condicional, es la centralización de la variación

Es importante aclarar que el problema no es usar condicionales, sino usarlos como mecanismo principal para manejar la variación.

Cuando todas las reglas están concentradas en un mismo método:

- Cada cambio obliga a modificar código existente.
- Se incrementa el riesgo de introducir errores colaterales.
- Se pierde claridad sobre qué hace realmente la clase.
- Se dificulta la reutilización de partes del sistema.

Desde una perspectiva de diseño, este patrón indica que el sistema no está preparado para el cambio, sino que reacciona a él de forma improvisada.

En el Capítulo 2 se estudió qué es una clase mal diseñada y cómo refactorizarla desde el punto de vista de responsabilidades, cohesión y acoplamiento.

En este capítulo, el foco es distinto: Aquí no se analiza una clase mal diseñada desde el inicio, sino una clase que se degrada a medida que el sistema crece. Es decir, una clase puede haber comenzado siendo razonable, pero termina convirtiéndose en una clase problemática por la forma en que se extiende. Este matiz es clave y representa el aporte central de este capítulo.

Ejemplo conceptual (sin código aún)

Imaginemos un sistema que inicialmente maneja solo:

- Tipo A

Luego se agrega:

- Tipo B

Después:

- Tipo C

Y más adelante:

- Tipo D

Cada nuevo tipo se implementa agregando una condición más. El sistema sigue funcionando, pero:

- El método principal se vuelve más largo.
- Las reglas se mezclan.
- El orden de los condicionales comienza a importar.
- Aparecen dependencias implícitas entre casos.

En este punto, el sistema ya no "crece", sino que se expande de forma desordenada.

Consecuencias técnicas de la explosión de condicionales

Desde una perspectiva técnica, este enfoque provoca:

1. Baja extensibilidad

   Cada nueva variante requiere modificar código existente, lo que contradice la idea de crecimiento controlado.

2. Dificultad para probar

   Probar un solo caso exige recorrer múltiples ramas del código.

3. Aumento del acoplamiento

El método central conoce demasiados detalles de cada caso.

4. Pérdida de legibilidad

   El flujo de ejecución se vuelve difícil de seguir incluso para quien escribió el código.

5. Riesgo de regresiones

   Cambiar una regla puede afectar silenciosamente a otras.

Señal temprana de alerta para el estudiante

Una señal clara de que se está cayendo en este error es cuando el estudiante se pregunta:

- "¿Dónde agrego este nuevo caso?"
- "¿En qué if va esto?"
- "¿Qué pasa si cambio esta condición?"

Estas preguntas indican que el diseño no está orientado a objetos, sino a casos.

Este capítulo introduce un cambio de mentalidad fundamental: En lugar de agregar más condiciones, se deben agregar más objetos. Cuando el sistema crece por variación, la solución orientada a objetos no consiste en ampliar decisiones, sino en delegar comportamientos.

Esto conduce naturalmente al uso consciente de:

- interfaces,
- clases especializadas,
- composición,
- polimorfismo aplicado al diseño.

Ejemplo real: Sistema de facturación de servicios domésticos

Una empresa factura servicios a domicilio: Plomería, Electricidad y Pintura.

Cada servicio calcula su costo de forma distinta:

- Plomería: mano de obra + materiales
- Electricidad: mano de obra + recargo por urgencia (si aplica)
- Pintura: mano de obra + costo por metro cuadrado

Diseño típico (malo) que "funciona" pero no es extensible

```
class FacturadorServicios {

    public double calcularCosto(String tipoServicio, double
manoObra, double materiales,
                                boolean urgente, double metrosCua-
drados) {

        if (tipoServicio.equalsIgnoreCase("Plomeria")) {
            return manoObra + materiales;

        } else if (tipoServicio.equalsIgnoreCase("Electricidad"))
{
            double total = manoObra;
            if (urgente) total += 15;
            return total;
```

```
        } else if (tipoServicio.equalsIgnoreCase("Pintura")) {
            return manoObra + (metrosCuadrados * 2.5);
        }

        return 0;
    }
}
```


¿Por qué este diseño es problemático? (justificación con Cap. 2)

1. Baja cohesión: la clase mezcla reglas distintas en un solo método.
2. Alto acoplamiento: el método conoce detalles de todos los servicios.
3. No extensible: al agregar "Carpintería", hay que modificar este método.
4. Difícil de probar: los casos se multiplican y se vuelven incontrolables.

En resumen: el sistema "funciona", pero crece mal.

El diseño limpio busca lo contrario: que cada nueva variación se agregue como una unidad separada, sin modificar el núcleo. Para lograrlo, necesitamos un mecanismo que encapsule las reglas que cambian.

Habiendo comprendido por qué la explosión de condicionales es un problema estructural y no solo estilístico, el siguiente paso es aprender cómo encapsular esas variaciones para que el sistema pueda crecer sin modificar lo que ya funciona. En la siguiente sección se mostrará cómo transformar una lógica basada en decisiones en un diseño basado en objetos con responsabilidades claras.

3.2. Encapsular lo que cambia: diseño por estrategia de comportamiento

Después de identificar el problema de la explosión de condicionales, surge una pregunta clave: ¿Cómo permitir que un sistema crezca sin convertir cada nuevo requisito en un nuevo if? La respuesta no está en agregar más lógica a una clase central, sino en cambiar la forma de pensar el diseño. En Programación Orientada a Objetos, la solución consiste en encapsular aquello que varía y aislarlo del resto del sistema.

La idea clave del diseño por variación

El principio fundamental que guía esta sección es: Si algo cambia con frecuencia, no debe estar incrustado dentro de una clase central; debe convertirse en una entidad independiente.

Esto implica aceptar que:

- los cambios no son excepciones,
- los cambios son normales,
- el diseño debe anticiparlos.

En muchos sistemas reales, lo que cambia no es el proceso general de registrar un pedido, sino las reglas específicas: cómo se calcula un costo, cómo se aplica un recargo, cómo se gestiona una excepción. Por ello, en lugar de escribir múltiples reglas dentro de un mismo método, se define un contrato común que represente ese comportamiento variable.

¿Qué significa realmente "encapsular lo que cambia"?

Encapsular lo que cambia no significa simplemente "crear más clases". Significa:

- Identificar con precisión qué parte del comportamiento es inestable.

- Separar esa parte en una unidad con una responsabilidad clara.
- Hacer que el resto del sistema dependa de una abstracción, no de una implementación concreta.

En términos prácticos, el sistema deja de preguntar:

- "¿Qué tipo es este?"
- "¿Qué caso debo ejecutar?"

y comienza a delegar:

- "Ejecuta tu comportamiento".

Este cambio reduce la complejidad cognitiva del diseño.

Del pensamiento condicional al pensamiento orientado a objetos

Un enfoque basado en condicionales responde a la pregunta:

- ¿Qué tengo que hacer si el tipo es X?

Un enfoque orientado a objetos responde a otra pregunta:

- ¿Quién debe saber cómo hacer esto?

Esta diferencia es sutil, pero profunda. En el primer caso, una clase central sabe demasiado. En el segundo, cada objeto sabe hacer lo que le corresponde.

Este es el punto donde la Programación Orientada a Objetos deja de ser solo una técnica de codificación y se convierte en una forma de modelar el problema.

El contrato común como punto de estabilidad

Cuando se define un contrato común (interfaz o clase abstracta), se establece un punto de estabilidad en el sistema. Este contrato:

- no cambia con cada nuevo requisito,
- define qué se espera del comportamiento,
- permite que nuevas variantes se incorporen sin modificar el código existente.

En el ejemplo del cálculo de costos, el contrato representa la idea general de "calcular un costo", sin imponer cómo debe hacerse. Cada variante concreta se ocupa de su propia lógica.

Esta separación crea una frontera clara entre:

- el proceso estable (registrar pedido, coordinar flujo),
- y el comportamiento variable (cálculo, recargo, descuento).

Una vez comprendido que el crecimiento desordenado del sistema se produce cuando la variación se concentra en una clase central, el siguiente paso lógico consiste en materializar la idea del contrato como punto de estabilidad.

En el problema que se analiza a continuación, se ha identificado con claridad que lo que cambia con mayor frecuencia es la forma de calcular el costo del servicio, no el proceso general de facturación. Por tanto, resulta incorrecto que una única clase conozca y gestione todas las reglas posibles. En su lugar, el diseño orientado a objetos propone extraer ese comportamiento variable y representarlo mediante una abstracción.

Esa abstracción actúa como un acuerdo común: define qué se debe hacer (calcular el costo de un servicio), pero no cómo se hace. De esta forma, el sistema puede crecer incorporando nuevas reglas de cálculo sin necesidad de modificar el código existente, sino simplemente añadiendo nuevas implementaciones que respeten dicho contrato.

A partir de este punto, el diseño se desarrollará de manera progresiva y guiada, mostrando cómo:

- se define el contrato común,
- se modelan los datos de entrada de forma cohesionada,
- se encapsulan las variaciones en clases independientes,
- y se mantiene un contexto estable que coordina el proceso.

Este enfoque elimina la explosión de condicionales y prepara el sistema para evolucionar de forma ordenada y mantenible, como se muestra en la siguiente implementación paso a paso.

La idea clave es: Si algo cambia con frecuencia, no lo escribas dentro de una clase central: conviértelo en una clase separada. En este caso, lo que cambia es el cálculo del costo. Por tanto, se define un contrato común.

- **Paso 1: Definir una interfaz para el cálculo**

```
interface CalculableServicio {
    double calcularCosto(TrabajoServicio trabajo);
}
```

- **Paso 2: Modelar los datos de entrada (alta cohesión)**

```
class TrabajoServicio {
    private double manoObra;
    private double materiales;
    private boolean urgente;
```

```
    private double metrosCuadrados;

    public TrabajoServicio(double manoObra, double materiales, boolean urgente, double metrosCuadrados) {
        this.manoObra = manoObra;
        this.materiales = materiales;
        this.urgente = urgente;
        this.metrosCuadrados = metrosCuadrados;
    }

    public double getManoObra() { return manoObra; }
    public double getMateriales() { return materiales; }
    public boolean isUrgente() { return urgente; }
    public double getMetrosCuadrados() { return metrosCuadrados; }
}
```

Justificación: Se evita pasar muchos parámetros sueltos. Se mejora cohesión y claridad.

- **Paso 3: Implementar cada servicio como una clase (variación aislada)**

```
class ServicioPlomeria implements CalculableServicio {
    @Override
    public double calcularCosto(TrabajoServicio trabajo) {
        return trabajo.getManoObra() + trabajo.getMateriales();
    }
```

```
}
class ServicioElectricidad implements CalculableServicio {
    @Override
    public double calcularCosto(TrabajoServicio trabajo) {
        double total = trabajo.getManoObra();
        if (trabajo.isUrgente()) total += 15;
        return total;
    }
}
class ServicioPintura implements CalculableServicio {
    @Override
    public double calcularCosto(TrabajoServicio trabajo) {
      return trabajo.getManoObra() + (trabajo.getMetrosCuadrados() * 2.5);
    }
}
```

- **Paso 4: Crear un "contexto" que permanezca estable**

```
class FacturadorServicios {

    public double facturar(CalculableServicio servicio, TrabajoServicio trabajo) {
        return servicio.calcularCosto(trabajo);
    }
}
```

Demostración (main)

```
public class MainServicios {

   public static void main(String[] args) {


      FacturadorServicios facturador = new FacturadorServicios();


     TrabajoServicio trabajo1 = new TrabajoServicio(30, 12, false, 0);

      double totalPlomeria = facturador.facturar(new ServicioPlomeria(), trabajo1);

      System.out.println("Plomería: $" + totalPlomeria);


      TrabajoServicio trabajo2 = new TrabajoServicio(40, 0, true, 0);

      double totalElectricidad = facturador.facturar(new ServicioElectricidad(), trabajo2);

       System.out.println("Electricidad urgente: $" + totalElectricidad);


      TrabajoServicio trabajo3 = new TrabajoServicio(25, 0, false, 20);

      double totalPintura = facturador.facturar(new ServicioPintura(), trabajo3);

      System.out.println("Pintura: $" + totalPintura);

   }

}
```

¿Qué ganamos con este diseño?

- Para agregar "Carpintería" solo creamos ServicioCarpinteria, sin tocar FacturadorServicios.
- El código está dividido en clases con alta cohesión.
- El acoplamiento se reduce: el facturador no conoce los detalles de cada servicio.

Ahora el estudiante ya ve una forma profesional de extender comportamiento. Sin embargo, surge una duda real: ¿siempre se debe usar interfaces? ¿cuándo conviene herencia? ¿y cuándo composición? La siguiente sección responde esa decisión con criterios concretos.

3.3. Herencia vs composición vs interfaces: guía de decisión con criterios de diseño

En el Capítulo 1 se estudiaron los conceptos fundamentales de la Programación Orientada a Objetos: herencia, polimorfismo, abstracción e interfaces. Allí se explicó qué son, cómo se declaran en Java y cómo se representan en UML. Sin embargo, conocer la sintaxis y la teoría no garantiza un buen diseño. En la práctica, uno de los mayores desafíos para el estudiante y también para desarrolladores con experiencia, no es saber cómo usar herencia o interfaces, sino decidir cuándo conviene usar cada una.

Este capítulo introduce precisamente ese nivel de madurez en el diseño: pasar del uso técnico de los mecanismos de la POO al uso consciente y justificado de los mismos.

La pregunta ya no es "¿puedo usar herencia?", sino "¿debo usar herencia aquí?"

Responder correctamente esta pregunta evita muchos de los problemas vistos en el Capítulo 2 y permite

construir sistemas extensibles sin recurrir a parches posteriores.

La decisión de diseño como eje del capítulo

En sistemas reales, una mala elección entre herencia, composición o interfaces suele provocar:

- jerarquías rígidas difíciles de modificar,
- duplicación de código innecesaria,
- clases con responsabilidades mezcladas,
- alto acoplamiento entre módulos.

Por ello, esta sección no se enfoca en repetir definiciones, sino en establecer criterios prácticos que ayuden al estudiante a tomar decisiones correctas desde el diseño inicial o durante la refactorización.

¿Cuándo conviene usar herencia?

La herencia es una herramienta poderosa, pero también una de las más mal utilizadas. En muchos casos, se emplea simplemente para reutilizar código, sin analizar si la relación entre clases es conceptualmente correcta. Esto conduce a jerarquías forzadas que dificultan la evolución del sistema.

La herencia conviene cuando se cumplen simultáneamente los siguientes criterios:

- Relación “es un” clara y estable.

La herencia debe representar una relación natural del dominio del problema. La subclase debe poder sustituir completamente a la clase base sin alterar el significado del programa.

Ejemplos conceptualmente correctos:

- Un EmpleadoDocente es un Empleado.

- Una CuentaAhorros es una Cuenta.
- Un LibroDigital es un Libro.

Ejemplos conceptualmente incorrectos:

- Un Pedido no es un Cliente.
- Un ReportePDF no es un ArchivoBaseDeDatos.
- Un PagoTarjeta no es un Pedido.

Si la frase “X es un Y” no suena natural en lenguaje cotidiano, la herencia probablemente no sea la mejor opción.

Comportamiento y estructura comunes reales

La herencia es apropiada cuando las clases hijas comparten atributos y comportamientos de forma genuina, y no solo de manera superficial. Esto significa que:

- la superclase define un comportamiento común significativo,
- las subclases solo especializan o ajustan ese comportamiento,
- no es necesario sobrescribir la mayoría de los métodos para “corregir” la herencia.

Si una subclase necesita anular constantemente métodos heredados porque “no aplican”, es una señal clara de mal uso de la herencia.

Jerarquía estable y no explosiva

La herencia funciona mejor cuando la jerarquía:

- es relativamente estable en el tiempo,
- no crece de forma impredecible,
- representa categorías claras del dominio.

Por ejemplo, una jerarquía de:

- tipos de transporte,
- tipos de cuentas bancarias,
- tipos de empleados,

suele ser más estable que una jerarquía de:

- promociones,
- reglas de negocio,
- métodos de cálculo,
- políticas de descuento.

Cuando se prevé que aparecerán muchas variantes nuevas, la herencia suele volverse rígida y difícil de mantener, lo que indica que composición o interfaces pueden ser una mejor alternativa.

Relación con cohesión y acoplamiento (Capítulo 2)

Desde los criterios estudiados en el Capítulo 2:

- Una herencia bien aplicada aumenta la cohesión, ya que agrupa comportamientos relacionados en una superclase.
- Una herencia mal aplicada incrementa el acoplamiento, porque obliga a las subclases a depender de decisiones de diseño que no les corresponden.

Por ello, la herencia debe verse como una decisión estructural fuerte, no como un atajo para reutilizar código.

Advertencia

Un error frecuente en estudiantes es pensar: “Si uso herencia, mi diseño es automáticamente orientado a

objetos". Esto no es cierto. Un diseño con herencia puede ser incluso peor que uno sin ella si:

- fuerza relaciones artificiales,
- mezcla responsabilidades,
- dificulta la extensión.

En muchos casos, no usar herencia es la mejor decisión de diseño.

Luego de analizar los riesgos de usar herencia de forma indiscriminada, es importante aclarar que la herencia no es incorrecta por sí misma. El problema no es la herramienta, sino el criterio con el que se utiliza. Existen escenarios en los que la herencia no solo es válida, sino que resulta la opción más natural y expresiva del modelo.

Uno de esos escenarios aparece cuando el dominio del problema presenta categorías claras, estables y naturalmente jerárquicas, donde la relación "es un" se cumple sin ambigüedad y se mantiene en el tiempo. En estos casos, la herencia permite representar el modelo de forma directa, legible y alineada con la realidad.

El sistema de transporte urbano constituye un ejemplo clásico de este tipo de situación. En él, conceptos como Bus, Metro y Taxi no son comportamientos intercambiables ni roles temporales, sino tipos bien definidos de un mismo concepto general: el transporte. En lenguaje cotidiano y técnico, se puede afirmar sin esfuerzo que:

- un Taxi es un Transporte,
- un Bus es un Transporte,
- un Metro es un Transporte.

Además, todos estos tipos comparten características estructurales (por ejemplo, identificación del vehículo) y un comportamiento común que debe especializarse (el cálculo de la tarifa). Esta combinación de estructura compartida y comportamiento común con variaciones específicas es precisamente el contexto en el que la herencia resulta adecuada.

Por esta razón, en el siguiente ejemplo se emplea una clase abstracta que representa el concepto general de transporte, y se derivan de ella clases concretas que implementan su propia lógica de cálculo de tarifa. Este diseño:

- refleja fielmente el dominio del problema,
- evita duplicación de código,
- mantiene alta cohesión,
- y no introduce rigidez innecesaria, ya que la jerarquía es estable y comprensible.

Con esta justificación clara, se presenta a continuación el modelo basado en herencia para el sistema de transporte urbano.

Ejemplo: Sistema de transporte urbano (Bus, Metro, Taxi son tipos de Transporte).

```
abstract class Transporte {
    protected String placa;

    public Transporte(String placa) {
        this.placa = placa;
    }
```

```
    public abstract double calcularTarifa(double km);
}
class Taxi extends Transporte {
    public Taxi(String placa) { super(placa); }

    @Override
    public double calcularTarifa(double km) {
        return 1.5 + (km * 0.8);
    }
}
class Bus extends Transporte {
    public Bus(String placa) { super(placa); }

    @Override
    public double calcularTarifa(double km) {
        return 0.35; // tarifa plana
    }
}
```

Justificación: La jerarquía representa un concepto real y estable. Alta cohesión y lectura clara.

¿Cuándo conviene usar composición?

Si la herencia responde a la pregunta "¿qué es este objeto?", la composición responde a una pregunta diferente y, en muchos sistemas reales, mucho más

frecuente: “¿de qué está compuesto este objeto?” o “¿qué capacidades utiliza este objeto?”.

La composición consiste en construir clases a partir de otras clases, delegando parte de su comportamiento en objetos internos, en lugar de heredarlo. Este enfoque favorece diseños más flexibles, menos rígidos y mejor preparados para el cambio, especialmente cuando el dominio del problema no presenta jerarquías estables.

En términos de diseño orientado a objetos moderno, la composición no es una alternativa “de segundo nivel”, sino una estrategia preferente en la mayoría de los sistemas extensibles.

Relación “tiene un”

Usa composición cuando la relación entre clases no describe identidad, sino posesión o uso. Un objeto no es otro objeto, sino que lo contiene, lo utiliza o depende de él para cumplir su función. Esta diferencia semántica es clave para un buen diseño.

Ejemplos conceptuales claros:

- Un Pedido tiene un método de pago (pero no es un método de pago).
- Un Vehículo tiene un motor (pero no es un motor).
- Un Usuario tiene una estrategia de autenticación (pero no es la autenticación).

Forzar estas relaciones en jerarquías de herencia suele producir modelos artificiales, difíciles de mantener y conceptualmente incorrectos. En estos casos, la composición refleja mejor la realidad y evita relaciones “forzadas” que no existen en el dominio.

Necesitas variar comportamiento sin heredar

La composición es especialmente poderosa cuando el comportamiento debe cambiar, pero el tipo del objeto no. Mientras que la herencia fija el comportamiento en tiempo de diseño (la subclase define cómo se comporta), la composición permite variar el comportamiento en tiempo de ejecución, simplemente cambiando el objeto interno que se utiliza.

Esto resulta fundamental cuando:

- el mismo objeto puede comportarse de distintas maneras según el contexto,
- el comportamiento no define la identidad del objeto,
- o diferentes combinaciones de comportamientos son posibles.

Por ejemplo, un sistema puede tener:

- múltiples formas de cálculo,
- diferentes reglas de descuento,
- varios métodos de pago,
- distintos algoritmos de validación.

Intentar modelar estas variaciones mediante herencia conduce rápidamente a jerarquías profundas, rígidas y difíciles de extender. En cambio, con composición, el objeto principal permanece estable y delegar el comportamiento se vuelve trivial.

El cambio es frecuente o combinable

Este es uno de los criterios más importantes del capítulo y conecta directamente con la idea central: el diseño orientado a objetos debe anticipar el cambio.

Cuando un comportamiento:

- cambia con frecuencia,
- se combina con otros comportamientos,
- se amplía continuamente con nuevas variantes,

la herencia se convierte en un obstáculo. Cada nuevo caso implica crear una nueva subclase o modificar la jerarquía existente, aumentando el acoplamiento y el riesgo de errores.

La composición, en cambio:

- aísla el cambio en clases pequeñas,
- permite agregar nuevas variantes sin tocar el código existente,
- y evita la “explosión de subclases”.

Este criterio enlaza directamente con lo visto en la sección anterior sobre explosión de condicionales: cuando ni los if ni la herencia escalan bien, la composición aparece como la solución natural.

Advertencia

Un error común en estudiantes es pensar que usar herencia siempre es “más orientado a objetos” que usar composición. En realidad, ocurre lo contrario: en sistemas reales, la mayoría de las relaciones son de composición, no de herencia. La herencia debe reservarse para conceptos nucleares, estables y claramente jerárquicos. La composición, en cambio, es la herramienta principal para modelar variación, flexibilidad y evolución del sistema. Por esta razón, los diseños modernos bien estructurados suelen presentar:

- pocas jerarquías de herencia, y
- muchas relaciones de composición bien definidas.

Con los criterios anteriores, ya no se trata únicamente de saber qué es herencia o qué es composición, sino de tomar decisiones de diseño conscientes. En particular, cuando el comportamiento no define la identidad del objeto, cuando puede cambiar con el tiempo o incluso variar entre instancias, la herencia deja de ser la herramienta adecuada.

En este tipo de situaciones, insistir en jerarquías conduce a diseños rígidos y artificiales. Por ejemplo, modelar una cuenta como CuentaBasica, CuentaAvanzada, CuentaPremiumConRecomendadorAvanzado, etc., genera una explosión de clases que no aporta claridad conceptual y dificulta la evolución del sistema. El problema no es el tipo de cuenta en sí, sino el comportamiento que varía.

Aquí es donde la composición muestra su verdadero valor: el objeto principal permanece estable, mientras que el comportamiento variable se encapsula en objetos especializados que pueden intercambiarse sin modificar la estructura central del sistema.

Este enfoque resulta especialmente natural en sistemas modernos como plataformas de streaming, donde funcionalidades como las recomendaciones, los planes de suscripción o las reglas de personalización cambian con frecuencia y evolucionan de forma independiente. En estos casos, una cuenta no es un recomendador, sino que utiliza uno para cumplir su función.

El siguiente ejemplo ilustra cómo aplicar composición para resolver este tipo de problema de manera limpia, flexible y extensible, evitando jerarquías forzadas y preparando el diseño para futuras variaciones.

Ejemplo: Aplicación de streaming

Un plan de suscripción tiene un tipo de recomendador (básico o avanzado).

```
interface Recomendador {
    String recomendar();
}

class RecomendadorBasico implements Recomendador {
    public String recomendar() { return "Recomendación basada en tendencias". ; }
}

class RecomendadorAvanzado implements Recomendador {
    public String recomendar() { return "Recomendación basada en tu historial". ; }
}

class CuentaStreaming {
    private Recomendador recomendador;

    public CuentaStreaming(Recomendador recomendador) {
        this.recomendador = recomendador;
    }

    public void mostrarRecomendacion() {
        System.out.println(recomendador.recomendar());
```

```
    }
}
```

Justificación: La cuenta no “es un recomendador”, la cuenta **tiene un recomendador.** Esto evita jerarquías forzadas.

¿Cuándo conviene usar interfaces?

Las interfaces son una de las herramientas más poderosas y a la vez más malinterpretadas de la Programación Orientada a Objetos. Muchos estudiantes las ven únicamente como un requisito técnico del lenguaje o como una alternativa a la herencia múltiple, cuando en realidad su valor principal está en definir contratos estables de comportamiento.

Usar una interfaz significa comprometerse a que una clase ofrecerá un conjunto de operaciones, sin imponer cómo deben implementarse ni qué estructura interna debe tener la clase. Por ello, las interfaces no describen qué es un objeto, sino qué sabe hacer.

La interfaz como contrato

Una interfaz representa un acuerdo: cualquier clase que la implemente garantiza que podrá responder a los métodos definidos en ella. Este contrato permite que otras partes del sistema trabajen con objetos sin conocer su clase concreta, favoreciendo el bajo acoplamiento y la extensibilidad.

Este enfoque resulta especialmente útil cuando:

- varias clases distintas necesitan ofrecer la misma funcionalidad,
- las clases no están relacionadas jerárquicamente,
- cuando el comportamiento puede combinarse de múltiples formas.

Por ejemplo, conceptos como pagable, exportable, notificable, autenticable o persistible no definen una jerarquía natural de clases. Son capacidades, no identidades.

Capacidades vs jerarquías

Una regla clave de diseño es la siguiente:

Si el concepto describe lo que un objeto puede hacer, y no lo que es, probablemente debe modelarse como una interfaz.

Mientras que la herencia responde a la pregunta "¿qué tipo de cosa es este objeto?", la interfaz responde a "¿qué capacidades ofrece este objeto?". Esta diferencia evita jerarquías forzadas y modelos poco expresivos.

Por ejemplo:

- Un Documento puede ser imprimible.
- Una Factura puede ser imprimible.
- Un Reporte puede ser imprimible.

Forzar una jerarquía común solo por compartir la acción de imprimir carece de sentido conceptual. En cambio, una interfaz Imprimible expresa claramente la intención del diseño.

Múltiples habilidades sin herencia múltiple

Otro beneficio clave de las interfaces es que una clase puede implementar varias interfaces, combinando distintas capacidades sin caer en los problemas de la herencia múltiple. Esto permite construir objetos ricos en comportamiento sin acoplarlos a jerarquías rígidas.

Por ejemplo, una misma clase podría ser:

- Autenticable,
- Notificable,

- Auditable.

Cada una de estas capacidades puede evolucionar de forma independiente, sin afectar la estructura principal de la clase.

Interfaces como puntos de estabilidad

En sistemas que crecen, las interfaces suelen actuar como puntos de estabilidad: mientras las implementaciones cambian, el contrato permanece. Esto reduce el impacto del cambio, facilita pruebas, permite simulaciones (mocking) y favorece la evolución del sistema sin reescribir código existente.

Desde una perspectiva de diseño profesional, las interfaces:

- reducen el acoplamiento,
- facilitan la extensión sin modificación,
- y mejoran la legibilidad del modelo al hacer explícitas las capacidades del sistema.

Síntesis conceptual

En términos prácticos, conviene usar interfaces cuando:

- Se necesita un contrato común para múltiples implementaciones.
- La funcionalidad se comparte por capacidad, no por jerarquía.
- Un objeto puede tener más de una habilidad al mismo tiempo.
- Se desea aislar lo que cambia de lo que permanece estable.

Una vez establecido el criterio de decisión para el uso de interfaces, resulta fundamental observarlo aplicado en un contexto concreto. El objetivo no es aprender una nueva sintaxis, sino reconocer el tipo de problema que exige una interfaz como solución natural.

En muchos sistemas reales, especialmente aquellos orientados a la generación de información, aparecen funcionalidades que deben ofrecerse en distintos formatos sin que exista entre ellos una relación jerárquica. Exportar información a PDF o a Excel no define qué es un objeto, sino qué puede hacer. Ambos formatos cumplen la misma finalidad, pero no comparten estructura, estado ni identidad conceptual.

Intentar modelar este escenario mediante herencia conduciría a una jerarquía artificial (por ejemplo, ExportadorPDF y ExportadorExcel heredando de una clase base concreta), cuando en realidad lo único que se necesita es garantizar que ambos objetos cumplan con un contrato común de exportación.

Las interfaces permiten expresar exactamente esta intención: distintas clases, posiblemente muy diferentes internamente, ofrecen una misma capacidad. De esta forma, el sistema puede trabajar con ellas de manera uniforme, sin conocer detalles de implementación ni forzar relaciones inexistentes.

El siguiente ejemplo ilustra cómo aplicar este criterio de diseño de forma clara y extensible, utilizando una interfaz para representar la capacidad de exportar reportes en distintos formatos.

Ejemplo: Sistema de exportación de reportes

```
interface Exportable {
    void exportar(String contenido);
}
```

```
class ExportadorPDF implements Exportable {
    public void exportar(String contenido) {
        System.out.println("Exportando a PDF: " + contenido);
    }
}

class ExportadorExcel implements Exportable {
    public void exportar(String contenido) {
        System.out.println("Exportando a Excel: " + contenido);
    }
}
```

Justificación: PDF y Excel no tienen por qué pertenecer a una jerarquía; comparten una capacidad.

Guía comparativa de decisiones de diseño

Hasta este punto, se han analizado tres mecanismos fundamentales de la Programación Orientada a Objetos: herencia, composición e interfaces. Más importante que conocer su definición es saber elegir correctamente entre ellos, ya que una mala decisión de diseño suele ser más costosa que un error de implementación.

La Tabla 3.1 resume los criterios fundamentales de decisión, sirviendo como guía práctica de consulta para el estudiante:

Tabla 3.1. Comparación entre Herencia, Composición e Interfaces.

Criterio	Herencia	Composición	Interfaces
Tipo de relación	"Es un"	"Tiene un"	"Puede hacer"
Define identidad	Sí	No	No
Define comportamiento	Sí (compartido y especializado)	Sí (delegado)	Sí (contrato)
Acoplamiento	Alto (jerarquía fija)	Bajo	Muy bajo
Flexibilidad	Baja a media	Alta	Muy alta
Permite múltiples combinaciones	No	Sí	Sí
Ideal cuando	La jerarquía es clara y estable	El comportamiento varía o se combina	La funcionalidad es una capacidad
Riesgo principal	Jerarquías rígidas	Diseño disperso si se abusa	Interfaces vacías o sin sentido
Ejemplo típico	Tipos de transporte	Recomendadores, estrategias	Exportadores, pagadores

Interpretación

No existe una única "mejor" solución de diseño. Un buen diseño orientado a objetos equilibra estas tres herramientas, eligiendo cada una según el contexto del problema y no por costumbre o preferencia personal. La herencia debe usarse con moderación y solo cuando la relación conceptual es fuerte y estable. La composición es la opción preferida cuando se necesita flexibilidad y variación de comportamiento. Las interfaces permiten construir sistemas extensibles basados en contratos, reduciendo dependencias y favoreciendo la evolución del software.

Con estos criterios claros, el estudiante está preparado para afrontar problemas reales de diseño, evitando soluciones improvisadas y justificando cada decisión desde una perspectiva técnica y conceptual.

Hasta este punto del capítulo se han establecido los criterios de decisión para elegir entre herencia, composición e interfaces. Sin embargo, comprender los criterios de forma aislada no garantiza que el estudiante sea capaz de aplicarlos correctamente en situaciones reales, donde los problemas no vienen etiquetados ni delimitados de forma clara.

En la práctica profesional, los sistemas se presentan como descripciones del mundo real, con restricciones, cambios frecuentes y ambigüedades. Por ello, antes de escribir una sola línea de código, es indispensable analizar el problema, identificar qué cambia, qué permanece estable y qué tipo de relación existe entre los elementos.

La siguiente sección se analizan casos de estudio concretos desde una perspectiva metodológica, es decir, se explica por qué una solución es adecuada y por qué las otras no lo son, evitando respuestas automáticas o mecánicas. El objetivo no es llegar rápido a una solución, sino entrenar el razonamiento de diseño.

3.4. Análisis metodológico de casos de estudio: decidir correctamente el diseño

El diseño orientado a objetos no consiste en aplicar herencia, composición o interfaces de forma automática, sino en tomar decisiones justificadas. Cada decisión incorrecta de diseño suele manifestarse más adelante como:

- rigidez ante cambios,

- duplicación de código,
- explosión de clases,
- alto acoplamiento.

En los siguientes casos de estudio se presenta una descripción del problema, seguida de un análisis conceptual, que conduce a la elección correcta del mecanismo de diseño. El énfasis no está en la implementación, sino en el razonamiento que conduce a la solución.

- **Caso 1: Tipos de empleados en una empresa**

Descripción del problema

Una empresa maneja distintos tipos de empleados: administrativos, técnicos y gerentes. Todos comparten información básica (nombre, identificación, salario base), pero cada tipo calcula su salario final de manera diferente según reglas propias.

Análisis metodológico

Aquí existe una relación conceptual clara y natural: un gerente es un empleado, no "tiene" un empleado ni "actúa como" empleado. La identidad del objeto está definida desde el dominio del problema y no cambia con el tiempo. El comportamiento variable (cálculo del salario) está directamente ligado al tipo de empleado, no a una estrategia intercambiable.

Decisión correcta: Herencia

La herencia permite:

- reutilizar atributos comunes,
- sobrescribir el cálculo del salario,
- mantener una jerarquía clara y estable.

Por qué no composición

Usar composición implicaría que un empleado "tenga" un tipo de empleado, lo cual es conceptualmente incorrecto y confuso.

Por qué no interfaces

Una interfaz no modela identidad. No tiene sentido decir que un gerente "implementa" empleado.

- **Caso 2: Sistema de notificaciones (correo, SMS, push)**

Descripción del problema

El sistema debe enviar notificaciones por diferentes canales, y estos canales pueden aumentar o cambiar con el tiempo.

Análisis metodológico

Correo, SMS y push no son tipos de una misma cosa, sino formas distintas de ejecutar una acción. No existe una jerarquía natural entre ellos. Lo único que comparten es la capacidad de enviar notificaciones. Además, el sistema debe poder agregar nuevos canales sin modificar código existente.

Decisión correcta: Interfaces

Por qué no herencia

Forzar una jerarquía implicaría una clase base artificial que no representa una identidad real.

Por qué no composición

Sin una interfaz, el sistema quedaría acoplado a implementaciones concretas.

- **Caso 3: Vehículo con motor intercambiable**

Descripción del problema

Un vehículo puede usar motor eléctrico, a combustión o híbrido. El tipo de motor puede variar incluso dentro del mismo modelo.

Análisis metodológico

El vehículo no es un motor. El motor es una parte del vehículo que puede cambiar sin que el vehículo deje de ser el mismo. Esto es un ejemplo clásico de relación "tiene un".

Decisión correcta: Composición

Por qué no herencia

Si el tipo de motor define la clase del vehículo, cambiar el motor implicaría cambiar la jerarquía completa.

Por qué no interfaces únicamente

El motor tiene estado y lógica propia; necesita ser un objeto concreto.

- **Caso 4: Formatos de exportación de reportes**

Descripción del problema

El sistema exporta reportes en PDF, Excel y CSV.

Análisis metodológico

PDF y Excel no comparten identidad ni estructura interna. Comparten únicamente la capacidad de exportar.

Decisión correcta: Interfaces

Por qué no herencia

No existe una jerarquía natural entre formatos.

Por qué no composición sin interfaz

El sistema quedaría rígido ante nuevas implementaciones.

- **Caso 5: Tipos de cuentas bancarias**

Descripción del problema

Existen cuentas de ahorro, corriente y empresarial, cada una con reglas propias.

Análisis metodológico

Las cuentas representan categorías estables del dominio bancario. Un cambio de tipo implica un cambio conceptual profundo.

Decisión correcta: Herencia

Por qué no composición

Una cuenta no "tiene" una cuenta de ahorro.

Por qué no interfaces

La cuenta necesita compartir estado y comportamiento base.

Casos de decisión mixta (diseño realista)

- **Caso 6: Plataforma educativa (usuarios + métodos de evaluación)**

Entidades y límites del problema: El sistema maneja Usuarios. Dentro de ese concepto aparecen dos categorías: Estudiante y Profesor. Además, el sistema aplica métodos de evaluación (por ejemplo: examen, proyecto, rúbrica, quiz, evaluación por pares), que pueden variar según asignatura, docente o periodo académico.

Identidad vs capacidad vs regla variable: Estudiante y Profesor no son "modos de funcionamiento" intercambiables; son roles identitarios del usuario en el dominio académico. En lenguaje natural: un estudiante es un usuario; un profesor es un usuario.

Los métodos de evaluación no definen qué es el usuario. Un estudiante no "es un examen" ni "es una

rúbrica". La evaluación es una política/regla que se aplica y puede cambiar.

Evaluación del cambio: Es razonable que los tipos de usuario (Estudiante/Profesor) sean estables y no se multipliquen sin control. En cambio, los métodos de evaluación sí tienden a crecer: aparecen variantes, combinaciones y cambios frecuentes (nuevas rúbricas, ponderaciones, evaluaciones híbridas).

Análisis comparativo de alternativas:

Alternativa A: Solo herencia (Usuario Estudiante/Profesor + Evaluación por herencia)

Esto lleva a diseños del tipo: EstudianteConExamen, EstudianteConProyecto, EstudianteConRúbrica, etc.

Si además hay variaciones del método (examen en línea vs presencial), la jerarquía crece aún más.

Consecuencia concreta: explosión de clases (combinaciones), rigidez y duplicación.

Alternativa B: Solo interfaces (Estudiante y Profesor como "habilidades")

Si Estudiante y Profesor fueran interfaces, se pierde la idea de una base común estructural de "usuario" y se vuelve confuso el modelo: ¿Dónde quedan atributos comunes obligatorios como identificación, credenciales, contacto, etc.?

Se diluye la identidad del dominio (usuario académico) en "capacidades sueltas".

Alternativa C: Herencia para identidad + composición para evaluación

Herencia modela lo estable (Usuario → Estudiante/Profesor).

Composición encapsula lo variable (estrategia/método de evaluación), permitiendo cambiarlo sin alterar la identidad.

Decisión justificada

Decisión correcta: Herencia + Composición.

Herencia representa correctamente la identidad del dominio (Estudiante/Profesor son Usuarios).

Composición evita explotar la jerarquía y permite que la evaluación cambie, se combine o evolucione, manteniendo bajo acoplamiento y alta cohesión.

- **Caso 7: Sistema de transporte urbano (Bus, Taxi + tarifas por ciudad)**

Entidades y límites del problema: Existen medios de transporte concretos: Bus y Taxi.

El sistema debe calcular tarifa según recorrido, pero las reglas cambian por ciudad (o incluso por zona, horario, subsidios, eventos).

Identidad vs capacidad vs regla variable

Bus y Taxi sí representan tipos del dominio: Bus es un Transporte; Taxi es un Transporte.

La tarifa, en cambio, es una política externa: no define qué es el bus o taxi, sino una regla dependiente del contexto (ciudad/municipio).

Evaluación del cambio

Es estable que existan pocos tipos de transporte (bus, taxi, quizá metro).

Pero las reglas tarifarias cambian con frecuencia: nuevas ordenanzas, tarifas nocturnas, recargos, promociones.

Análisis comparativo de alternativas

Alternativa A: Solo herencia (cada ciudad como subclase)

Se crearían clases como:

TaxiQuito, TaxiGuayaquil, BusQuito, BusGuayaquil, etc.

Consecuencia concreta: explosión de clases por producto cartesiano: (tipo de transporte) × (ciudad).

Cada nueva ciudad o cambio tarifario obliga a tocar jerarquías o crear más clases.

Alternativa B: Condicionales dentro de calcularTarifa

Un switch(ciudad) dentro del método.

Consecuencia concreta: vuelves al problema central del capítulo: explosión de condicionales, baja extensibilidad y alto riesgo de regresiones.

Alternativa C: Herencia para tipo + contrato (interfaces) para política de tarifa

Herencia representa identidad: Bus/Taxi como transportes.

Una interfaz (p. ej. PoliticaTarifa) representa el contrato estable para "cómo se calcula tarifa en un contexto".

Decisión justificada

Decisión correcta: Herencia + Interfaces.

Herencia para el "qué es" (bus/taxi).

Interfaz para el "cómo se calcula en este contexto" (tarifa por ciudad), permitiendo múltiples implementaciones sin jerarquías artificiales ni condicionales gigantes.

- **Caso 8: Aplicación bancaria moderna (cuenta + notificación/auditoría/seguridad)**

Entidades y límites del problema: Existe un concepto central: Cuenta (ahorro/corriente/empresa). Además,

se requieren funcionalidades transversales: Notificar, Auditar, Aplicar seguridad (doble factor, bloqueo por intentos, alertas).

Identidad vs capacidad vs regla variable

La cuenta sigue siendo “Cuenta”.

Notificación, auditoría y seguridad son capacidades/servicios, no “tipos de cuenta”.

Evaluación del cambio

Estas capacidades cambian mucho:

- nuevos canales de notificación,
- nuevas políticas de auditoría,
- nuevas reglas de seguridad por normativa o amenazas.

Análisis comparativo de alternativas

Alternativa A: Herencia para combinar capacidades

Clases como: Cuenta Con Notificacion, Cuenta Con Auditoria, Cuenta Con Seguridad, Cuenta Con Notificacion Y Auditoria, etc.

Consecuencia concreta: explosión combinatoria. Además, una cuenta puede requerir varias capacidades simultáneas y variables por configuración.

Alternativa B: Todo dentro de la clase Cuenta

Métodos enviarAlerta(), registrarAuditoria(), validarSeguridad() dentro de Cuenta.

Consecuencia concreta: Cuenta se convierte en clase “Dios”, pierde cohesión, aumenta acoplamiento y cada cambio de seguridad obliga a modificar Cuenta.

Alternativa C: Composición de servicios + interfaces para contratos

Cuenta “tiene” un servicio de notificación, “tiene” un auditor, “tiene” una política de seguridad.

Interfaces definen contratos (Notificador, Auditor, PoliticaSeguridad) y permiten múltiples implementaciones.

Decisión justificada

Decisión correcta: Composición + Interfaces.

Porque el problema no pide nuevos “tipos” de cuenta por cada capacidad; pide capacidades combinables, extensibles y reemplazables sin alterar la identidad de la cuenta.

- **Caso 9: Sistema hospitalario (personal médico + turnos/roles)**

Entidades y límites del problema: El hospital gestiona personal: Médico y Enfermero. Además, gestiona turnos (diurno/nocturno/guardia) y roles temporales (jefe de guardia, responsable de área, suplente).

Identidad vs capacidad vs regla variable

Médico y Enfermero son identidades del dominio: son tipos de personal.

Turnos y roles son atributos o comportamientos que cambian. Un médico no deja de ser médico por cambiar de turno.

Evaluación del cambio

Los tipos de personal suelen ser estables.

Los turnos/roles cambian por calendario, necesidades, reemplazos; pueden combinarse.

Análisis comparativo de alternativas

Alternativa A: Herencia para turnos/roles

Clases como MedicoNocturno, MedicoDiurno, MedicoJefeGuardia, etc.

Consecuencia concreta: jerarquía explosiva e irreal. Los roles son temporales; no deben rigidizarse en clases.

Alternativa B: Solo composición sin jerarquía de personal

Si todo fuera "Personal con componentes", se pierde el beneficio de identidad: validaciones, permisos, responsabilidades propias del tipo (médico prescribe, enfermero administra).

Alternativa C: Herencia para identidad + composición para turnos/roles

Herencia define estructura/atributos comunes del personal y especialización.

Composición permite asignar turnos/roles como objetos o configuraciones intercambiables.

Decisión justificada

Decisión correcta: Herencia + Composición.

Representa correctamente el dominio: identidad estable + condiciones operativas variables sin jerarquía artificial.

- **Caso 10: Plataforma de comercio electrónico (productos + pagos + entregas)**

Entidades y límites del problema: La tienda vende Productos físicos (requieren envío), digitales (descarga/licencia). Además, gestiona:

1. Pagos (tarjeta, transferencia, billetera),

2. Entrega (courier, retiro, envío express, descarga inmediata).

Identidad vs capacidad vs regla variable

Producto físico y digital sí son tipos del dominio (identidad).

Pago y entrega son capacidades/estrategias: se eligen, cambian y se amplían con el tiempo.

Evaluación del cambio

Los tipos de producto suelen ser pocos y estables.

Los métodos de pago y entrega crecen constantemente y dependen de integraciones externas.

Análisis comparativo de alternativas

Alternativa A: Solo herencia para todo

Crear ProductoFisico con PagoTarjeta con EntregaExpress, etc.

Consecuencia concreta: explosión combinatoria inmediata.

Alternativa B: Condicionales en un "ProcesadorPedido" central

if (tipoProducto == FISICO) ... if (pago == TARJETA) ... if (entrega == EXPRESS)...

Consecuencia concreta: explosión de condicionales (problema 3.2), alto acoplamiento y dificultad de pruebas.

Alternativa C: Herencia para producto + interfaces y composición para pago/entrega

Herencia separa identidad (físico vs digital) y comportamiento estable del producto.

Pago y entrega se definen por contratos (interfaces) y se inyectan/seleccionan (composición).

Decisión justificada

Decisión correcta: Herencia + Composición + Interfaces.

Porque el dominio exige modelar tanto identidad (tipos de producto) como variación extensible (métodos de pago y entrega) sin caer en jerarquías forzadas ni condicionales masivos.

A lo largo de este capítulo se han establecido y analizado de manera sistemática los criterios de decisión que permiten elegir entre herencia, composición e interfaces en el diseño orientado a objetos. Mediante el estudio comparativo de casos reales y mixtos, se ha demostrado que estas decisiones no responden a reglas rígidas, sino al análisis del dominio del problema, la identificación de identidades, comportamientos y capacidades, así como a la evaluación del impacto del cambio sobre la arquitectura del sistema.

Sin embargo, el verdadero dominio de estos conceptos se alcanza cuando el estudiante es capaz de integrarlos en un único sistema, donde coexisten múltiples entidades, reglas variables y extensiones futuras. En este contexto, el desafío ya no consiste en aplicar un solo mecanismo de diseño, sino en combinar adecuadamente herencia, composición e interfaces, manteniendo altos niveles de cohesión y bajo acoplamiento, incluso frente al crecimiento progresivo de los requisitos.

Con este propósito, se presenta a continuación un caso integrador, concebido como un escenario realista de evolución del software. El caso permitirá analizar cómo un sistema puede ampliarse mediante variaciones funcionales sin necesidad de romper su estructura

original, justificando cada decisión de diseño desde una perspectiva metodológica y evaluando sus efectos sobre la mantenibilidad, extensibilidad y calidad del código.

3.5 Caso integrador del capítulo: "Panadería El Buen Pan": crecimiento por variaciones sin romper el sistema

Propósito del caso: aplicar lo aprendido en el Cap. 2 (responsabilidades, refactorización, cohesión y acoplamiento) para resolver un problema típico de crecimiento: aparecen nuevas variantes (entregas, pagos, comprobantes) y el sistema debe extenderse sin volverse un caos.

3.5.1 Contexto real del problema

Una panadería vende productos (pan, pasteles, café, etc.). El sistema debe registrar pedidos y soportar variaciones reales:

1. Tipos de entrega

- Mostrador (no tiene costo de envío)
- Domicilio (costo de envío depende de distancia)
- Programada (se agenda una fecha/hora; recargo por agenda)

2. Métodos de pago

- Efectivo (puede aplicar descuento si paga exacto)
- Tarjeta (recargo por comisión)
- Transferencia (requiere comprobante/código)

3. Comprobante

- Ticket simple
- Factura (requiere datos del cliente: RUC/CI, dirección)

En la vida real, estos cambios aparecen con el tiempo. El error típico es “parchear” agregando condicionales y haciendo crecer una clase central hasta convertirla en una clase Dios.

Fase A — Diseño inicial (malo pero común) y sus problemas

Primera versión: clase monolítica con condicionales

Esta versión “funciona”, pero crece mal. Se presenta para que el estudiante aprenda a detectar el problema.


```
import java.util.*;


class SistemaPanaderia {


    public void registrarPedido(String clienteNombre, String
clienteId, String direccion,
                                List<String> productos, List<Double>
precios,
                                String tipoEntrega, double distanciaKm,
Date fechaProgramada,
                                String metodoPago, boolean pagaExacto,
                                boolean requiereFactura) {


        // 1) Calcular subtotal
        double subtotal = 0;
        for (double p : precios) subtotal += p;


        // 2) Calcular entrega
```

```
        double costoEntrega = 0;
        if (tipoEntrega.equalsIgnoreCase("Mostrador")) {
            costoEntrega = 0;
        } else if (tipoEntrega.equalsIgnoreCase("Domicilio")) {
            costoEntrega = 1.5 + (distanciaKm * 0.3);
      } else if (tipoEntrega.equalsIgnoreCase("Programada"))
{
            costoEntrega = 2.0; // recargo fijo
            if (fechaProgramada == null) {
                System.out.println("Error: entrega programada
requiere fecha". );
                return;
            }
        }

        // 3) Calcular total preliminar
        double total = subtotal + costoEntrega;

        // 4) Aplicar reglas de pago
        if (metodoPago.equalsIgnoreCase("Efectivo")) {
            if (pagaExacto) total -= 0.5; // descuento
        } else if (metodoPago.equalsIgnoreCase("Tarjeta")) {
            total *= 1.03; // comisión 3%
        } else if (metodoPago.equalsIgnoreCase("Transferen-
cia")) {
```

```
        // requiere comprobante (simulado)
        System.out.println("Transferencia: recuerde enviar comprobante". );
    }

    // 5) Generar comprobante
    if (!requiereFactura) {
        System.out.println("=== TICKET ===");
        System.out.println("Cliente: " + clienteNombre);
        System.out.println("Productos: " + productos);
        System.out.println("Subtotal: $" + subtotal);
        System.out.println("Entrega: $" + costoEntrega);
        System.out.println("Total: $" + total);
        System.out.println("==============");
    } else {
        if (clienteId == null || direccion == null) {
            System.out.println("Error: factura requiere identificación y dirección". );
            return;
        }
        System.out.println("=== FACTURA ===");
        System.out.println("Cliente: " + clienteNombre + " (" + clienteId + ")");
        System.out.println("Dirección: " + direccion);
        System.out.println("Productos: " + productos);
```

```
            System.out.println("Subtotal: $" + subtotal);
            System.out.println("Entrega: $" + costoEntrega);
            System.out.println("Total: $" + total);
            System.out.println("===============");
        }

        System.out.println("Pedido registrado con éxito". );
    }
}
```


Análisis de lo realizado (por qué este diseño es deficiente)

Aplicando criterios del Capítulo 2:

1. Baja cohesión (la clase hace demasiadas cosas)

   Esta clase:

   - calcula subtotal,
   - calcula costo de entrega,
   - valida reglas de entrega,
   - aplica reglas de pago,
   - genera comprobantes,
   - valida requisitos de factura,
   - imprime salidas.

Está concentrando múltiples responsabilidades distintas.

2. Alto acoplamiento (mezcla decisiones de negocio con salida)

Si cambia:

- la fórmula de domicilio,
- la comisión de tarjeta,
- el formato del ticket,
- el requisito de factura,

entonces hay que modificar el mismo método. Esto provoca dependencia fuerte y fragilidad.

1. "Explosión de condicionales" (mal escalamiento)

   Cada nuevo tipo (ej.: "Entrega express", "Pago QR", "Factura electrónica") agrega más if/else. El método crece y se vuelve difícil de probar.

2. Reutilización baja y pruebas difíciles

- No se puede probar el cálculo de entrega sin ejecutar todo el flujo.
- No hay piezas independientes para testear.

Conclusión: esta solución funciona, pero no es extensible ni mantenible. Es la semilla de una clase Dios.

Fase B — Refactorización guiada: de mal diseño a diseño limpio (paso a paso)

Objetivo: mantener el comportamiento, pero mejorar estructura con:

1. alta cohesión,
2. bajo acoplamiento,
3. variaciones encapsuladas.

- **Paso 1: Crear modelos del dominio (clases con propósito claro)**

Primero se crean clases que representen el problema: Cliente, Producto, Pedido.

```
class Producto {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public String getNombre() { return nombre; }
    public double getPrecio() { return precio; }
}

class Cliente {
    private String nombre;
     private String identificacion; // CI/RUC (opcional según comprobante)
   private String direccion;     // opcional según comprobante

    public Cliente(String nombre, String identificacion, String direccion) {
        this.nombre = nombre;
        this.identificacion = identificacion;
```

```
        this.direccion = direccion;
    }

    public String getNombre() { return nombre; }
    public String getIdentificacion() { return identificacion; }
    public String getDireccion() { return direccion; }
}

import java.util.List;

class Pedido {
    private Cliente cliente;
    private List<Producto> productos;

    public Pedido(Cliente cliente, List<Producto> productos) {
        this.cliente = cliente;
        this.productos = productos;
    }

    public Cliente getCliente() { return cliente; }
    public List<Producto> getProductos() { return productos; }

    public double calcularSubtotal() {
        double subtotal = 0;
```

```
        for (Producto p : productos) subtotal += p.getPrecio();
        return subtotal;
    }
}
```

Justificación:

1. Cada clase tiene una responsabilidad clara (alta cohesión).
2. El subtotal ahora pertenece naturalmente al Pedido, no a un método gigante.

- **Paso 2: Encapsular la variación "Entrega" (interfaz + implementaciones)**

Aquí aparece el primer principio de este capítulo: encapsular lo que cambia.

```
interface Entrega {
    double calcularCosto(Pedido pedido);
    String descripcion();
}
class EntregaMostrador implements Entrega {
    public double calcularCosto(Pedido pedido) { return 0; }
    public String descripcion() { return "Mostrador"; }
}
class EntregaDomicilio implements Entrega {
    private double distanciaKm;

    public EntregaDomicilio(double distanciaKm) {
```

```
        this.distanciaKm = distanciaKm;
    }

    public double calcularCosto(Pedido pedido) {
        return 1.5 + (distanciaKm * 0.3);
    }

    public String descripcion() { return "Domicilio (" + distanciaKm + " km)"; }
}
import java.util.Date;

class EntregaProgramada implements Entrega {
    private Date fechaProgramada;

    public EntregaProgramada(Date fechaProgramada) {
        this.fechaProgramada = fechaProgramada;
    }

    public double calcularCosto(Pedido pedido) {
        return 2.0; // recargo
    }

    public String descripcion() { return "Programada (" + fechaProgramada + ")"; }
}
```

Justificación:

1. Ya no hay if/else para entregas.
2. Para agregar “EntregaExpress”, se crea una nueva clase sin modificar el sistema.

- **Paso 3: Encapsular la variación “Pago”**

```
interface MetodoPago {
    double aplicarReglas(double total);
    String descripcion();
}

class PagoEfectivo implements MetodoPago {
    private boolean pagaExacto;

    public PagoEfectivo(boolean pagaExacto) {
        this.pagaExacto = pagaExacto;
    }

    public double aplicarReglas(double total) {
        return pagaExacto ? total - 0.5 : total;
    }

    public String descripcion() { return “Efectivo”; }
}

class PagoTarjeta implements MetodoPago {
```

```
    public double aplicarReglas(double total) {
        return total * 1.03;
    }
    public String descripcion() { return "Tarjeta (+3%)"; }
}

class PagoTransferencia implements MetodoPago {
    public double aplicarReglas(double total) {
        System.out.println("Transferencia: recuerde enviar
comprobante". );
        return total;
    }
    public String descripcion() { return "Transferencia"; }
}
```

Justificación:

1. Cada método de pago es un módulo independiente.
2. Evita crecer con condicionales y mantiene cohesión.

- **Paso 4: Encapsular la variación "Comprobante"**

```
interface Comprobante {
    void emitir(Pedido pedido, Entrega entrega, MetodoPago
pago, double totalFinal);
}

class Ticket implements Comprobante {
```

```
    public void emitir(Pedido pedido, Entrega entrega, Metodo
Pago pago, double totalFinal) {
        System.out.println("=== TICKET ===");
          System.out.println("Cliente: " + pedido.getCliente().
getNombre());
        System.out.println("Productos:");
        for (Producto p : pedido.getProductos()) {
              System.out.println("- " + p.getNombre() + " $" +
p.getPrecio());
        }
       System.out.println("Entrega: " + entrega.descripcion());
        System.out.println("Pago: " + pago.descripcion());
        System.out.println("Total: $" + totalFinal);
        System.out.println("==============");
    }
}

class Factura implements Comprobante {

    public void emitir(Pedido pedido, Entrega entrega, Metodo
Pago pago, double totalFinal) {

        Cliente c = pedido.getCliente();
          if (c.getIdentificacion() == null || c.getDireccion() ==
null) {
          System.out.println("Error: factura requiere identifica-
```

```
ción y dirección". );
            return;
        }

        System.out.println("=== FACTURA ===");
        System.out.println("Cliente: " + c.getNombre() + " (" +
c.getIdentificacion() + ")");
        System.out.println("Dirección: " + c.getDireccion());
        System.out.println("Productos:");
        for (Producto p : pedido.getProductos()) {
            System.out.println("- " + p.getNombre() + " $" +
p.getPrecio());
        }
        System.out.println("Entrega: " + entrega.descripcion());
        System.out.println("Pago: " + pago.descripcion());
        System.out.println("Total: $" + totalFinal);
        System.out.println("==============");
    }
}
```

Justificación:

1. Cada comprobante gestiona su propio formato y validaciones.
2. Se evita que "Pedido" o un "SistemaPanadería" se conviertan en clase Dios.

- **Paso 5: Crear el coordinador (flujo estable del proceso)**

Ahora sí: un gestor con responsabilidad clara, parecido al patrón que ya se practicó en Capítulo 2.

```
class GestorPedidos {

    public void registrar(Pedido pedido, Entrega entrega, MetodoPago pago, Comprobante comprobante) {

        double subtotal = pedido.calcularSubtotal();
        double costoEntrega = entrega.calcularCosto(pedido);

        double total = subtotal + costoEntrega;
        double totalFinal = pago.aplicarReglas(total);

        comprobante.emitir(pedido, entrega, pago, totalFinal);

        System.out.println("Pedido registrado con éxito.\n");
    }
}
```

Justificación:

1. Alta cohesión: el gestor coordina el proceso.
2. Bajo acoplamiento: trabaja con interfaces (Entrega, MétodoPago, Comprobante).
3. Extensible: se agregan nuevas variaciones sin cambiar el gestor.

Demostración completa (Main) con diferentes combinaciones

```
import java.util.*;

public class MainPanaderia {

    public static void main(String[] args) {

        // Productos
        List<Producto> productos = Arrays.asList(
                new Producto("Pan de queso", 0.50),
                new Producto("Café", 1.00),
                new Producto("Pastel pequeño", 1.75)
        );

        // Cliente con datos completos (para factura)
                Cliente cliente1 = new Cliente("Luis Andrade",
"0923456789", "Av. Central y 10 de Agosto");

        // Pedido
        Pedido pedido1 = new Pedido(cliente1, productos);

        // Variaciones
        Entrega entrega1 = new EntregaDomicilio(4.0);
        MetodoPago pago1 = new PagoTarjeta();
```

```
        Comprobante comp1 = new Factura();

        // Gestor
        GestorPedidos gestor = new GestorPedidos();
        gestor.registrar(pedido1, entrega1, pago1, comp1);

        // Otro caso: ticket + mostrador + efectivo exacto
        Cliente cliente2 = new Cliente("Ana Vera", null, null);
        Pedido pedido2 = new Pedido(cliente2, Arrays.asList(
                new Producto("Pan integral", 0.60),
                new Producto("Jugo", 1.20)
        ));

        Entrega entrega2 = new EntregaMostrador();
        MetodoPago pago2 = new PagoEfectivo(true);
        Comprobante comp2 = new Ticket();

        gestor.registrar(pedido2, entrega2, pago2, comp2);
    }
}
```


Análisis final de lo realizado (con criterios del Cap. 2)

1. ¿Se eliminó la clase Dios?

Sí. Ya no existe una clase que mezcle:

- cálculo,

- validación,
- formatos,
- pagos,
- entregas.

Las responsabilidades están distribuidas y cada módulo es pequeño.

2. Cohesión

- Pedido calcula subtotal: coherente con su responsabilidad.
- EntregaDomicilio solo calcula costo de domicilio: cohesión alta.
- PagoTarjeta solo aplica reglas de comisión: cohesión alta.
- Factura solo gestiona emisión de factura: cohesión alta.
- GestorPedidos solo coordina: cohesión alta.

3. Acoplamiento

GestorPedidos no depende de clases concretas, depende de interfaces.

Se puede agregar:

- EntregaExpress,
- PagoQR,
- FacturaElectronica,

sin modificar GestorPedidos.

4. Extensibilidad real

Este diseño permite una extensión “limpia”:

- Se agregan clases nuevas, no se reescriben las existentes.
- Se reduce el riesgo de errores por cambios.

Este caso integrador demuestra que la Programación Orientada a Objetos no se trata solo de crear clases, sino de diseñar sistemas que puedan evolucionar sin perder claridad. Al encapsular las variaciones (entrega, pago, comprobante) y mantener un flujo estable coordinado por un gestor, se logra un diseño con alta cohesión, bajo acoplamiento y extensibilidad real.

A lo largo de este capítulo se ha demostrado que la calidad de un sistema orientado a objetos no depende principalmente de su tamaño, sino de las decisiones de diseño que gobiernan su capacidad de cambio. Un sistema no se vuelve complejo por crecer, sino por crecer sin una estructura que anticipe la variación. Cuando las responsabilidades están bien distribuidas, las identidades correctamente modeladas y los comportamientos variables encapsulados, el crecimiento deja de ser una amenaza y se convierte en una evolución controlada.

Mediante el análisis de criterios de decisión y casos de estudio progresivamente más complejos, el estudiante ha aprendido a distinguir entre qué debe representarse mediante herencia, qué debe resolverse mediante composición y qué debe expresarse a través de interfaces como contratos de comportamiento. Esta capacidad de decisión metodológica permite evitar errores comunes como jerarquías rígidas, clases monolíticas, explosión de condicionales y acoplamientos innecesarios, todos ellos responsables de sistemas frágiles y difíciles de mantener.

El caso integrador presentado al final del capítulo ha puesto de manifiesto que un diseño orientado a objetos bien concebido no busca anticipar todos los

cambios posibles, sino crear una estructura flexible que pueda absorberlos sin romperse. La alta cohesión, el bajo acoplamiento y la correcta separación entre identidad, comportamiento y variación se consolidan, así como principios prácticos, aplicables y verificables en sistemas reales.

Con estas bases, el estudiante se encuentra preparado para avanzar hacia estructuras de mayor nivel de abstracción. En el siguiente capítulo se profundizará en cómo estas decisiones de diseño se reflejan en mecanismos más avanzados del lenguaje y del paradigma, tales como el uso de colecciones polimórficas, genericidad y la introducción de patrones de diseño básicos, herramientas que permiten generalizar soluciones, aumentar la reutilización del código y construir arquitecturas más robustas y expresivas.

04

Patrones de diseño básicos: soluciones estructuradas para sistemas en evolución

4.1. El patrón Strategy: control explícito de la variación del comportamiento

En los capítulos anteriores se ha construido progresivamente una base sólida para el diseño orientado a objetos desde una perspectiva conceptual, metodológica y aplicada. A partir de este recorrido, el estudiante ya cuenta con los elementos necesarios para comprender cómo diseñar clases correctamente y cómo estructurar relaciones entre ellas. Sin embargo, en el desarrollo de sistemas reales, estas decisiones no se toman de forma aislada, sino que se repiten de manera recurrente frente a problemas similares: variación de comportamiento, creación de objetos, notificación de cambios, extensión de funcionalidades o control de recursos compartidos. Cuando estas situaciones se abordan de manera improvisada, el resultado suele ser un diseño frágil, difícil de mantener y propenso a errores conforme el sistema crece.

Es en este punto donde cobran relevancia los patrones de diseño. Lejos de ser soluciones rígidas o fórmulas cerradas, los patrones de diseño constituyen estructuras probadas que encapsulan buenas prácticas de diseño orientado a objetos frente a problemas recurrentes. Su valor no radica en memorizar su nombre o su diagrama, sino en comprender qué problema resuelven, por qué existen y en qué contexto deben aplicarse. En este sentido, los patrones pueden entenderse como una formalización de las decisiones correctas de diseño estudiadas en los capítulos anteriores, elevándolas a un nivel de mayor abstracción y reutilización conceptual.

Este capítulo se centra en el estudio de patrones de diseño básicos, seleccionados por su alto valor formativo y su frecuencia de uso en sistemas profesionales. Cada patrón será abordado no como una definición teórica, sino como una respuesta estructurada a un problema real, partiendo de un diseño ingenuo, analizando sus

limitaciones y evolucionando hacia una solución más flexible y mantenible. De este modo, se refuerza la idea de que los patrones no introducen complejidad innecesaria, sino que organizan la complejidad inherente al cambio.

Al finalizar este capítulo, el estudiante no solo será capaz de reconocer y aplicar patrones de diseño como Strategy, Factory Method, Observer, Decorator o Singleton, sino que también habrá desarrollado un criterio más maduro para evaluar cuándo su uso es pertinente y cuándo una solución más simple resulta suficiente. Este enfoque crítico y reflexivo constituye un paso fundamental hacia la construcción de software robusto, extensible y alineado con estándares profesionales de calidad.

En el contexto del diseño orientado a objetos, el término diseño ingenuo no tiene una connotación peyorativa ni implica falta de inteligencia o capacidad técnica del desarrollador. Por el contrario, hace referencia a un diseño inicial, directo y funcional, que surge de una comprensión básica del problema, pero que no considera aún la evolución del sistema ni los principios de diseño a mediano y largo plazo.

Un diseño ingenuo se caracteriza por centrarse en resolver el problema inmediato, priorizando que el sistema "funcione", sin analizar en profundidad cómo ese diseño se comportará frente a cambios, extensiones o nuevas necesidades del dominio.

En el contexto de plataformas educativas apoyadas por inteligencia artificial, las decisiones de diseño orientadas a objetos y el uso adecuado de patrones arquitectónicos resultan determinantes para garantizar sistemas escalables, mantenibles y adaptables a distintos escenarios de aprendizaje. Investigaciones recientes destacan que la calidad del diseño de software constituye un factor clave en la integración

efectiva de la inteligencia artificial en entornos educativos contemporáneos (Chávez-Cárdena et al., 2025).

Uno de los problemas más recurrentes en el desarrollo de software no es la complejidad inicial de un sistema, sino su evolución. A medida que un sistema crece, aparecen nuevas reglas, excepciones, políticas de negocio y variantes de comportamiento que no estaban previstas en el diseño original. Cuando estas variaciones no se gestionan adecuadamente, el código tiende a degradarse, volviéndose rígido, difícil de mantener y propenso a errores.

Desde el punto de vista del diseño orientado a objetos, este fenómeno ocurre cuando el comportamiento que cambia con frecuencia se encuentra incrustado dentro de clases centrales, obligando a modificarlas cada vez que surge una nueva variante. El resultado habitual es la proliferación de estructuras condicionales, la violación del principio de responsabilidad única y la pérdida progresiva de claridad del diseño.

El patrón Strategy surge precisamente como una respuesta a este problema. Su propósito no es introducir complejidad innecesaria, sino aislar aquello que varía, permitiendo que el resto del sistema permanezca estable frente al cambio.

Considérese el caso de una empresa de mensajería que debe calcular el costo de envío de paquetes. Inicialmente, el sistema distingue tres tipos de envío:

- Envío local.
- Envío nacional.
- Envío internacional.

Cada tipo de envío posee reglas de cálculo distintas, basadas en factores como el peso, la distancia, recargos

administrativos o impuestos. Además, estas reglas no son estables en el tiempo: cambian por promociones, ajustes comerciales o regulaciones externas.

Desde el análisis del dominio, se observa que:

- El concepto de envío es estable.
- El algoritmo de cálculo del costo es altamente variable.

Este contraste es clave para el diseño.

Un enfoque frecuente, especialmente en etapas tempranas de aprendizaje, consiste en centralizar todas las reglas en una única clase mediante estructuras condicionales. Conceptualmente, esta solución parece razonable, ya que concentra el cálculo en un solo lugar. Sin embargo, desde una perspectiva de diseño, introduce problemas graves.

En este tipo de diseño:

- Cada nuevo tipo de envío obliga a modificar la clase existente.
- La clase asume múltiples responsabilidades.
- El código se vuelve progresivamente más difícil de leer y probar.
- El sistema queda cerrado al cambio, pero abierto al error.

Aunque funcional en el corto plazo, este enfoque no escala y contradice los principios de diseño analizados en los capítulos anteriores, particularmente aquellos relacionados con cohesión, acoplamiento y apertura al cambio.

El patrón Strategy propone un cambio conceptual fundamental: eliminar la variación del interior de la clase central y encapsularla en entidades independientes.

Desde esta perspectiva, el cálculo del costo de envío no es responsabilidad del sistema principal, sino de estrategias intercambiables que representan distintas formas de realizar dicho cálculo.

En términos prácticos, esto implica:

- Definir un contrato común que represente el comportamiento variable.
- Implementar cada variante en una clase independiente.
- Delegar la ejecución del comportamiento a la estrategia correspondiente.

El sistema deja de preguntar "*¿qué tipo eres?*" y pasa a decir "*ejecuta tu comportamiento*".

Ejemplo ilustrativo

Para comprender Strategy sin entrar en detalles de aún en código, conviene observar un caso que todos entienden: un mismo proceso operacional con reglas que varían según el contexto.

Imagine una empresa que gestiona tres modalidades de envío: local, nacional e internacional. Aunque cambien las reglas, el proceso general siempre es parecido:

1. El cliente solicita un envío.
2. Se registra el paquete (peso, dimensiones, destino).
3. Se calcula el costo.
4. Se informa al cliente el total.

El paso crítico es el (3): calcular el costo. Aquí aparecen las variaciones:

- Envío local: se cobra principalmente por peso, sin recargos complejos.

- Envío nacional: se cobra por peso + recargo fijo por manejo o distancia media.
- Envío internacional: se cobra por peso + recargos por aduana, trámites, seguros o zona.

Ahora, observe lo siguiente:

El sistema tiene una operación estable ("calcular costo"), pero las reglas internas cambian según el tipo. Esto significa que en la lógica del sistema existe un "punto de variación".

¿Cómo se detecta que Strategy es necesario?

Un estudiante o desarrollador puede darse cuenta de que debe usar Strategy cuando aparecen patrones como:

1. La misma operación se repite con variantes
2. "calcularCosto" se escribe una vez, pero con tres versiones "internas".
3. Aparecen condicionales para elegir la regla
4. "si es local…, si es nacional…, si es internacional…".
5. El negocio anuncia cambios frecuentes
6. promociones, recargos, cambios de tarifas.
7. Se prevé crecimiento
8. hoy hay 3 tipos, mañana puede haber 6, ampliando el sistema: express, nocturno, con seguro, etc.
9. El "tipo" se vuelve más importante que la entidad

> la clase central comienza a estar dominada por la lógica de selección de reglas.

¿Por qué el diseño mejora con Strategy?

Cuando se aplica Strategy, ocurre un cambio mental (y estructural) en el diseño:

- En lugar de una clase que decide qué regla aplicar, se construyen reglas encapsuladas.
- Se reemplaza la idea de “el sistema contiene todas las fórmulas” por “el sistema usa una fórmula seleccionada”.
- El sistema principal deja de cambiar cuando cambian las reglas.

En otras palabras, Strategy logra lo siguiente:

- Reduce acoplamiento: el sistema principal no depende de reglas concretas.
- Aumenta cohesión: cada regla vive en una unidad clara (una estrategia).
- Mejora mantenibilidad: cambiar una regla no implica tocar el resto.
- Facilita pruebas: cada estrategia se valida aisladamente.
- Evita degradación del diseño: no crecen condicionales con cada nueva variante.

Esta explicación prepara al lector para el punto clave: la variación debe salir de la clase central y convertirse en “módulos de comportamiento”.

Con esta intuición ya construida, podemos formalizar el patrón en código con mayor claridad.

Formalización del diseño

En esta sección se traduce la lógica anterior a una estructura de clases. La intención no es “escribir más código”, sino reorganizarlo para que el sistema pueda crecer sin romperse.

Paso 1: Identificar el punto de variación

Antes de programar, el estudiante debe aprender a formular esta pregunta:

¿Qué parte del sistema cambia más frecuentemente y por qué?

En el dominio de estudio, el proceso general de facturación del envío es estable, pero el algoritmo de cálculo del costo no lo es. Por tanto, el punto de variación es el cálculo del costo del envío. Este paso es metodológicamente crucial, si el estudiante no identifica el punto de variación, tenderá a colocar "todo" dentro de una clase central y el patrón no tendrá sentido.

Paso 2: Convertir la variación en un contrato estable

Una vez identificado lo variable, se define un "contrato" que representa lo que el sistema necesita sin imponer cómo se hace.

```
interface EstrategiaEnvio {

    double calcularCosto(double peso);

}
```

¿Qué evidencia este paso?

- Se está separando "la necesidad" (calcular costo) de "la implementación".
- El sistema se compromete a pedir siempre lo mismo: calcularCosto(...).
- A partir de aquí, todas las variantes quedan obligadas a respetar el mismo contrato.

¿Por qué era necesario?

Sin un contrato, el sistema dependería directamente de clases concretas. El contrato es el mecanismo que "desacopla" el contexto del detalle.

Paso 3: Encapsular cada regla en una estrategia concreta

Cada tipo de envío se modela como una clase separada que implementa el contrato:


```
class EnvioLocal implements EstrategiaEnvio {
    @Override
    public double calcularCosto(double peso) {
        return peso * 1.2;
    }
}

class EnvioNacional implements EstrategiaEnvio {
    @Override
    public double calcularCosto(double peso) {
        return peso * 2.5;
    }
}

class EnvioInternacional implements EstrategiaEnvio {
    @Override
    public double calcularCosto(double peso) {
        return (peso * 5.0) + 10;
```

```
    }
}
```

¿Qué evidencia este paso?

- La lógica queda aislada por variante.
- Cada estrategia tiene una sola razón de cambio: su propia regla.

Mejora directa del diseño

- Ya no se modifican condicionales.
- Se agregan nuevas reglas creando nuevas estrategias.

Metodológicamente, aquí el estudiante debe entender que una estrategia es una "política" de comportamiento.

Paso 4: Crear un contexto estable que delegue el cálculo

El contexto es la clase que "necesita" el cálculo, pero no debe conocer la regla interna. Finalmente, el sistema que utiliza estas estrategias se mantiene completamente ajeno a las reglas específicas de cálculo. Su única responsabilidad es delegar el trabajo:

```
class CalculadoraEnvio {

    private EstrategiaEnvio estrategia;

    public CalculadoraEnvio(EstrategiaEnvio estrategia) {
        this.estrategia = estrategia;
    }
```

```
    public double calcular(double peso) {
        return estrategia.calcularCosto(peso);
    }
}
```

¿Qué evidencia este paso?

- La calculadora no sabe si el envío es local, nacional o internacional.
- Solo sabe que existe una estrategia que puede calcular.

¿Por qué esto es clave?

- La estabilidad del sistema queda en CalculadoraEnvio.
- La variación queda en implementaciones de EstrategiaEnvio.

Este paso es el que materializa el principio aprendido en capítulos anteriores: Composición para variar comportamientos, en lugar de herencia rígida o condicionales.

Paso 5: Selección de estrategia en el uso (control del cambio en tiempo de ejecución)

Finalmente, el sistema utiliza la estrategia adecuada:

```
public class Main {
    public static void main(String[] args) {

        CalculadoraEnvio calcLocal =
            new CalculadoraEnvio(new EnvioLocal());
```

```
        System.out.println(calcLocal.calcular(10));

        CalculadoraEnvio calcInter =
            new CalculadoraEnvio(new EnvioInternacional());

        System.out.println(calcInter.calcular(10));
    }
}
```


Evidencia

- El cambio no implica “editar” CalculadoraEnvio.
- Solo implica “seleccionar” otra estrategia.

Esto es exactamente lo que se busca en diseño profesional: cambiar configuración en lugar de cambiar código central.

El estudiante debe llevarse una regla práctica que pueda aplicar en otros dominios: Si en un método aparecen condicionales que eligen entre varias fórmulas/reglas, y esas reglas pueden crecer o cambiar, entonces la variación debe encapsularse como estrategias.

Strategy no es un “patrón elegante”, sino un mecanismo de control del cambio que evita la degradación del diseño conforme el sistema crece.

4.2. Evaluación del diseño resultante

Una vez aplicado el patrón Strategy, no basta con afirmar que “el diseño mejora”. En un contexto formativo universitario, el estudiante debe ser capaz de evaluar el resultado con criterios claros y compararlo con el diseño inicial basado en condicionales. Esta evaluación

permite consolidar el criterio de diseño y evita que Strategy se perciba como una "técnica opcional" o un "estilo personal".

A continuación, se analizan ambos enfoques (mal diseño vs. diseño con Strategy) a través de dimensiones esenciales: cohesión, acoplamiento, extensibilidad, mantenibilidad, pruebas y legibilidad. El objetivo es que el lector pueda justificar por qué Strategy no solo "ordena" el código, sino que incrementa la calidad del sistema.

A. Cohesión: qué tan bien delimitadas están las responsabilidades

En el mal diseño (uso de condicionales): La clase central (por ejemplo, CalculadoraEnvio) tiende a asumir dos responsabilidades simultáneas:

1. Mantener el proceso general de cálculo (recibir peso/datos, producir un valor).
2. Contener y seleccionar todas las reglas de negocio (local, nacional, internacional).

Esto disminuye la cohesión porque la clase se convierte en un contenedor de "muchas reglas", es decir, su propósito deja de ser único y comienza a mezclar lógica operativa con lógica de política de negocio.

Con Strategy:

La cohesión aumenta porque el sistema queda distribuido de forma natural:

- El contexto (CalculadoraEnvio) mantiene una sola responsabilidad: delegar el cálculo.
- Cada estrategia (EnvioLocal, EnvioNacional, EnvioInternacional) mantiene una sola responsabilidad: calcular el costo según una regla.

El estudiante puede observar que Strategy mejora cohesión porque el sistema se divide en unidades con propósito claro. Esto se alinea con el Capítulo 2: las clases "buenas" se caracterizan por tener una responsabilidad principal y ser comprensibles como unidad.

B. Acoplamiento: dependencia entre partes del sistema

En el mal diseño (uso de condicionales): La clase central conoce explícitamente todas las variantes. Aunque el código no mencione directamente new EnvioLocal(), sí "conoce" la lógica concreta de cada tipo a través de condicionales. En la práctica, está acoplada a:

- El conjunto de tipos posibles.
- Las reglas específicas de cada tipo.
- El orden en que se evalúan los casos.
- Las decisiones internas del algoritmo.

Esto implica que cambiar una regla, o agregar una nueva, obliga a tocar la clase central. Por tanto, el acoplamiento es alto.

Con Strategy:

El acoplamiento se reduce porque CalculadoraEnvio depende únicamente del contrato EstrategiaEnvio. El contexto ya no conoce detalles. El diseño logra:

- Dependencia hacia abstracción (contrato) en vez de implementación.
- Capacidad de reemplazar estrategias sin alterar el contexto.

El estudiante debe apropiarse de la idea de que el "bajo acoplamiento" no significa "no depender de nada", sino depender de contratos estables, no de decisiones concretas.

C. Extensibilidad: agregar nuevas variantes sin romper el diseño

En el mal diseño (uso de condicionales)

Agregar un nuevo tipo de envío (por ejemplo, “Envío Express”) implica:

- Modificar el bloque condicional.
- Insertar una nueva rama.
- Riesgo de afectar ramas existentes.
- Riesgo de errores por orden de condiciones o casos no cubiertos.

Este tipo de crecimiento es acumulativo: cada variante aumenta el tamaño del método central y la complejidad de lectura.

Con Strategy:

Agregar una nueva variante implica:

1. Crear una nueva clase que implemente EstrategiaEnvio.
2. Probarla de forma aislada.
3. Seleccionarla donde corresponda.

No se modifica el contexto ni el código ya probado. El sistema crece “por adición”, no por modificación.

El estudiante debe aprender a distinguir entre:

- crecer modificando (más riesgo),
- crecer agregando (más control).

Strategy habilita el segundo enfoque.

D. Mantenibilidad: costo real del cambio

En el mal diseño (uso de condicionales)

Si cambia una regla de cálculo (por ejemplo, nuevo recargo internacional), el cambio ocurre dentro de la clase central. Esto genera efectos colaterales:

- Se tocan líneas que conviven con otras reglas.
- Cambios pequeños aumentan el riesgo de introducir defectos.
- El método se vuelve difícil de entender con el tiempo.

Además, se pierde la trazabilidad: "¿Dónde está la regla internacional?" puede estar mezclada con otras ramas y excepciones.

Con Strategy:

La regla internacional vive en su propia clase. El cambio es:

- Localizado,
- aislado,
- fácilmente revisable,
- con bajo riesgo de afectar otras reglas.

El mantenimiento no es solo "cambiar código", sino hacerlo con mínima probabilidad de efectos secundarios. Strategy reduce ese riesgo mediante encapsulación.

E. Pruebas: posibilidad real de validar el sistema

En el mal diseño (uso de condicionales)

Probar la regla internacional implica ejecutar el método central y garantizar que se ingrese por la rama correcta. Esto puede requerir:

- Configurar múltiples condiciones.
- Cubrir combinaciones de entrada.

- Probar casos que dependen de decisiones internas.

En sistemas reales, esto se agrava cuando las reglas usan más variables (zona, urgencia, peso volumétrico, seguros), y el método central crece.

Con Strategy:

Cada estrategia se prueba como una unidad aislada:

- entradas y salida esperada.
- sin depender del resto del sistema.
- sin necesidad de activar condiciones.

Esto facilita pruebas unitarias reales, alineadas con prácticas profesionales.

Un diseño testeable no es un lujo: es requisito para software confiable. Strategy incrementa la testabilidad por aislamiento.

F. Legibilidad y comprensión del diseño

En el mal diseño (uso de condicionales)

La lectura obliga a recorrer ramas y comprender casos. El método central se convierte en el "mapa" del sistema, pero es un mapa lleno de excepciones. El lector aprende a "seguir condiciones", no a "comprender diseño".

Con Strategy:

La lectura se vuelve estructural:

- existe un contrato (EstrategiaEnvio),
- existen implementaciones,
- el contexto delega.

El diseño se comprende por la organización de clases, no por la exploración de condicionales.

El estudiante aprende a leer software como arquitectura, no como lista de reglas.

El patrón Strategy no introduce nuevas capacidades al sistema; introduce orden frente a la variación. Si bien el patrón Strategy resuelve de manera eficaz el problema de cómo variar el comportamiento, existen escenarios en los que el desafío principal no radica en el comportamiento, sino en la creación de objetos complejos sin acoplar el código a clases concretas. En estos casos, el problema se desplaza desde el cómo se comporta un objeto hacia cómo se construye. Este escenario será abordado en la siguiente sección mediante el estudio del patrón Factory Method, que introduce un enfoque estructurado para desacoplar la creación de objetos del código que los utiliza.

4.3. El patrón Factory Method: creación desacoplada de objetos

En secciones anteriores se ha puesto énfasis en la correcta distribución de responsabilidades y en el control del cambio mediante abstracciones, composición y encapsulación del comportamiento. Sin embargo, existe un aspecto del diseño orientado a objetos que con frecuencia se subestima en etapas tempranas de aprendizaje: la creación de objetos.

Para muchos estudiantes, instanciar objetos es un acto trivial: se utiliza el operador new allí donde se necesita un objeto y se continúa con el flujo del programa. Este enfoque es funcional en programas pequeños, pero en sistemas reales introduce una dependencia silenciosa y peligrosa: el código de negocio queda acoplado a clases concretas.

Desde el punto de vista del diseño, esto plantea una cuestión fundamental: ¿Debe una clase conocer exactamente qué tipo concreto de objeto está creando,

o solo debería conocer qué tipo de comportamiento espera de ese objeto?

Cuando la creación de objetos se dispersa por el sistema y se realiza de manera directa, el crecimiento del software se vuelve problemático. Cada nuevo tipo, variante o configuración obliga a modificar código existente, violando principios de diseño ya estudiados, como el principio Abierto/Cerrado.

El patrón Factory Method surge como una respuesta estructurada a este problema, proponiendo un mecanismo para desacoplar el uso de un objeto de su proceso de creación.

Antes de introducir código, resulta útil observar el problema desde una perspectiva conceptual.

Supóngase un sistema encargado de emitir comprobantes de venta. El proceso general siempre es el mismo:

1. Se solicita la emisión del comprobante.
2. Se valida el monto.
3. Se genera el documento.
4. Se entrega el resultado.

Sin embargo, el tipo de comprobante puede variar:

- Factura (con impuestos).
- Boleta (sin impuestos).
- Nota de crédito (ajuste negativo).
- En el futuro: factura electrónica, comprobante internacional, etc.

Desde el punto de vista del dominio:

- El proceso de emisión es estable.
- El tipo de comprobante es variable.

Un diseño ingenuo tiende a mezclar ambas cosas, obligando al sistema a "decidir" qué objeto crear cada vez que se emite un comprobante. Esta decisión suele materializarse mediante condicionales o lógica distribuida, lo que incrementa la complejidad del sistema.

Aquí aparece una señal clara de que Factory Method es necesario:

- Cuando una clase crea objetos de distintos tipos según el contexto.
- Cuando esa decisión puede crecer o cambiar con el tiempo.
- Cuando el resto del sistema no debería verse afectado por esos cambios.

Un enfoque común consiste en centralizar la creación dentro de una clase, utilizando estructuras condicionales:

```
class EmisorComprobantes {


  public Comprobante emitir(String tipo, double monto) {
    if (tipo.equals("FACTURA")) {
      return new Factura(monto);
    } else if (tipo.equals("BOLETA")) {
      return new Boleta(monto);
    } else if (tipo.equals("NOTA_CREDITO")) {
      return new NotaCredito(monto);
    }
    return null;
```

```
        }
    }
```

Este diseño presenta múltiples problemas estructurales:

- Alto acoplamiento: la clase depende explícitamente de todas las clases concretas.
- Baja extensibilidad: agregar un nuevo comprobante implica modificar esta clase.
- Violación del principio Abierto/Cerrado: el código no está cerrado a modificaciones.
- Dificultad de mantenimiento: la lógica de creación crece de forma acumulativa.
- Pérdida de claridad conceptual: la clase mezcla emisión y decisión de tipo.

Desde una perspectiva metodológica, este diseño es un ejemplo claro de cómo un sistema comienza a degradarse no por su tamaño, sino por la forma en que se gestiona la variabilidad.

El patrón Factory Method propone una reorganización del diseño basada en un principio simple, el código que usa un objeto no debería ser responsable de decidir qué clase concreta instanciar.

En lugar de centralizar la creación mediante condicionales, el patrón introduce una jerarquía de fábricas, donde:

- Una clase base define el proceso general.
- Las subclases deciden qué objeto concreto crear.
- El sistema trabaja siempre con abstracciones.

De esta manera, la variación en los tipos de objetos se controla mediante extensión, no mediante modificación.

Formalización del diseño: pasos metodológicos

Paso 1: Identificar la abstracción común

El primer paso consiste en reconocer que todos los comprobantes comparten una identidad conceptual. Esta identidad se modela mediante una clase abstracta:

```
abstract class Comprobante {
    protected double monto;

    public Comprobante(double monto) {
        this.monto = monto;
    }

    public abstract double calcularTotal();
    public abstract String obtenerDescripcion();
}
```

Este paso es fundamental porque:

- Permite tratar todos los comprobantes de forma uniforme.
- Define un contrato estable para el resto del sistema.
- Evita que el código cliente dependa de implementaciones concretas.

Paso 2: Implementar las variantes concretas

Cada tipo de comprobante encapsula su lógica específica:

```
class Factura extends Comprobante {
```

```
    public Factura(double monto) {
        super(monto);
    }

    @Override
    public double calcularTotal() {
        return monto * 1.12;
    }

    @Override
    public String obtenerDescripcion() {
        return "Factura con IVA";
    }
}

class Boleta extends Comprobante {

    public Boleta(double monto) {
        super(monto);
    }

    @Override
    public double calcularTotal() {
        return monto;
```

```
    }

    @Override
    public String obtenerDescripcion() {
        return "Boleta sin impuestos";
    }
}

class NotaCredito extends Comprobante {

    public NotaCredito(double monto) {
        super(monto);
    }

    @Override
    public double calcularTotal() {
        return -monto;
    }

    @Override
    public String obtenerDescripcion() {
        return "Nota de crédito";
    }
}
```

Cada clase tiene una única razón de cambio, lo que incrementa la cohesión del sistema.

Paso 3: Definir la fábrica abstracta (núcleo del patrón)

La fábrica abstracta encapsula el proceso general de emisión y delega la creación concreta:

```
abstract class FabricaComprobantes {


    public Comprobante emitir(double monto) {
        Comprobante c = crearComprobante(monto);
      System.out.println("Emitido: " + c.obtenerDescripcion());
        return c;
    }


    protected abstract Comprobante crearComprobante(dou-
ble monto);

}
```

Aquí se evidencia el patrón:

- El método público define el flujo estable.
- El método abstracto representa el punto de variación.

Paso 4: Implementar fábricas concretas

```
class FabricaFacturas extends FabricaComprobantes {

    @Override

    protected Comprobante crearComprobante(double mon-
to) {

        return new Factura(monto);
```

```
    }
}
class FabricaBoletas extends FabricaComprobantes {
    @Override
    protected Comprobante crearComprobante(double monto) {
        return new Boleta(monto);
    }
}
class FabricaNotasCredito extends FabricaComprobantes {
    @Override
    protected Comprobante crearComprobante(double monto) {
        return new NotaCredito(monto);
    }
}
```

Cada nueva variante se agrega creando una nueva subclase, sin tocar código existente.

Paso 5: Uso desacoplado del sistema

```
public class Main {
    public static void main(String[] args) {

        FabricaComprobantes fabrica = new FabricaFacturas();
        Comprobante c1 = fabrica.emitir(100);
        System.out.println(c1.calcularTotal());
```

```
        fabrica = new FabricaBoletas();
        Comprobante c2 = fabrica.emitir(80);
        System.out.println(c2.calcularTotal());
    }
}
```


El código cliente no conoce las clases concretas; solo trabaja con abstracciones.

Factory Method enseña al estudiante que:

- la creación es una decisión de diseño,
- el operador new debe usarse con criterio,
- la herencia puede servir para extender comportamiento de creación, no solo para reutilizar código.

Strategy resolvió el problema de cómo se comportan los objetos; Factory Method resuelve el problema de cómo se crean. Ambos patrones comparten una filosofía común: aislar la variación y proteger la estabilidad del sistema central.

No obstante, en muchos sistemas reales, el desafío no es solo crear o comportarse correctamente, sino reaccionar a cambios que ocurren en otras partes del sistema. Cuando múltiples componentes deben ser notificados ante una modificación de estado, surge un nuevo problema de diseño.

Este escenario será abordado en la siguiente sección mediante el estudio del patrón Observer, que permite establecer relaciones de dependencia controladas entre objetos sin introducir acoplamiento rígido.

4.4. El patrón Observer: reacción estructurada a cambios de estado

A medida que un sistema de software crece, los objetos dejan de actuar de forma aislada y comienzan a interactuar de manera más intensa. Cambios en el estado de una parte del sistema suelen tener impacto en otras: una actualización de datos, una acción del usuario, una transacción completada o un evento externo pueden requerir que múltiples componentes reaccionen de forma coordinada.

Un diseño ingenuo suele resolver esta necesidad mediante llamadas directas entre objetos. El objeto que cambia de estado notifica explícitamente a todos los demás componentes que dependen de él. Aunque este enfoque funciona en sistemas pequeños, introduce rápidamente un problema estructural: el objeto que cambia se vuelve dependiente de todos los que reaccionan.

Desde el punto de vista del diseño orientado a objetos, este escenario plantea una dificultad clave: ¿Cómo permitir que múltiples objetos reaccionen a un cambio sin que el objeto que cambia conozca a todos sus dependientes?

El patrón Observer surge para resolver este problema, proporcionando un mecanismo estructurado de notificación que mantiene bajo el acoplamiento y permite la evolución del sistema sin degradar su diseño.

Ejemplo ilustrativo sin código: "Un cambio, múltiples reacciones"

Considérese un sistema de gestión académica. Cuando un estudiante actualiza su información personal, pueden ocurrir varias acciones derivadas:

- El sistema debe actualizar la base de datos.

- El módulo de notificaciones debe enviar un correo de confirmación.
- El módulo de estadísticas debe registrar el cambio.
- El módulo de auditoría debe almacenar el evento.

Desde el punto de vista del dominio, todas estas acciones dependen de un mismo evento: el cambio de datos del estudiante. Sin embargo, sería un error de diseño que la clase Estudiante conociera y llamara explícitamente a cada uno de estos módulos.

Aquí aparecen señales claras de que el patrón Observer es necesario:

- Un objeto cambia de estado.
- Existen múltiples reacciones posibles.
- Las reacciones pueden crecer o modificarse con el tiempo.
- El objeto que cambia no debería modificarse cada vez que aparece una nueva reacción.

Observer permite modelar este escenario de forma natural: el objeto que cambia emite una notificación, y los interesados deciden reaccionar o no.

Diseño ingenuo y sus problemas estructurales

Un diseño común consiste en codificar explícitamente las notificaciones dentro del objeto que cambia:

```
class Estudiante {


    public void actualizarDatos() {
        // actualizar datos
        notificarCorreo();
```

```
        actualizarEstadisticas();
        registrarAuditoria();
    }

    private void notificarCorreo() {}
    private void actualizarEstadisticas() {}
    private void registrarAuditoria() {}
}
```

Este diseño presenta problemas evidentes:

- Alto acoplamiento: la clase conoce a todos los módulos dependientes.
- Baja cohesión: la clase mezcla gestión de datos con notificaciones y auditoría.
- Escasa extensibilidad: agregar una nueva reacción implica modificar esta clase.
- Fragilidad: cualquier cambio puede afectar múltiples funcionalidades.

Metodológicamente, este diseño viola principios ya estudiados: responsabilidad única, bajo acoplamiento y apertura al cambio.

El patrón Observer propone una separación clara entre:

- el sujeto (objeto que cambia de estado),
- y los observadores (objetos que reaccionan al cambio).

La idea central es:

Definir una dependencia uno-a-muchos entre objetos, de modo que cuando el sujeto cambia de estado, todos

sus observadores son notificados automáticamente, sin que el sujeto conozca detalles concretos de ellos.

El sujeto no ejecuta acciones específicas; simplemente notifica. Cada observador decide qué hacer ante la notificación.

Formalización del diseño: pasos metodológicos

Paso 1: Definir el contrato del observador

El primer paso consiste en formalizar qué significa "reaccionar a un cambio". Esto se logra mediante una interfaz:

```
interface Observador {
    void actualizar();
}
```

Este contrato establece un punto de estabilidad: cualquier objeto que quiera reaccionar a cambios debe implementar este método.

Paso 2: Definir el sujeto observable

El sujeto mantiene una lista de observadores y se encarga de notificarlos cuando su estado cambia:

```
interface Sujeto {
    void agregarObservador(Observador o);
    void eliminarObservador(Observador o);
    void notificarObservadores();
}
```

Esta abstracción permite que el sistema dependa de contratos y no de implementaciones concretas.

Paso 3: Implementar el sujeto concreto

```
class Estudiante implements Sujeto {

    private List<Observador> observadores = new
ArrayList<>();

  @Override
  public void agregarObservador(Observador o) {
    observadores.add(o);
  }

  @Override
  public void eliminarObservador(Observador o) {
    observadores.remove(o);
  }

  @Override
  public void notificarObservadores() {
    for (Observador o : observadores) {
      o.actualizar();
    }
  }

  public void actualizarDatos() {
```

```
        // cambio de estado
        notificarObservadores();
    }
}
```

El objeto Estudiante no conoce qué hacen los observadores; solo sabe que debe notificarlos.

Paso 4: Implementar observadores concretos

```
class NotificadorCorreo implements Observador {
    @Override
    public void actualizar() {
        System.out.println("Enviando correo de confirmación");
    }
}
class RegistroAuditoria implements Observador {
    @Override
    public void actualizar() {
        System.out.println("Registrando auditoría");
    }
}
```

Cada observador encapsula una reacción específica.

Paso 5: Uso del patrón

```
public class Main {
    public static void main(String[] args) {
```

```
        Estudiante estudiante = new Estudiante();

        estudiante.agregarObservador(new NotificadorCorreo());
        estudiante.agregarObservador(new RegistroAuditoria());

        estudiante.actualizarDatos();
    }
}
```

El sistema permite agregar o eliminar reacciones sin modificar la clase Estudiante.

Valor metodológico

El patrón Observer enseña al estudiante que:

- los cambios de estado generan eventos,
- los eventos no deben acoplarse a reacciones concretas,
- la comunicación desacoplada es clave en sistemas escalables,
- un objeto no debe "saber quién lo escucha".

Este patrón es fundamental en arquitecturas modernas orientadas a eventos, interfaces gráficas, sistemas distribuidos y aplicaciones reactivas.

Observer complementa a Strategy y Factory Method al abordar un tercer tipo de variación: la reacción a cambios. Mientras Strategy controla cómo se ejecuta un comportamiento y Factory Method controla cómo

se crean los objetos, Observer controla quién debe reaccionar cuando algo ocurre.

En la siguiente sección se estudiará el patrón Decorator, que aborda un problema diferente pero igualmente frecuente: cómo extender funcionalidades de un objeto de manera dinámica sin recurrir a herencia rígida ni explosión de clases.

4.5. El patrón Decorator: extensión dinámica de funcionalidades sin herencia rígida

En el desarrollo de software orientado a objetos, es habitual que un sistema deba ampliar funcionalidades existentes sin alterar su estructura original. Estas ampliaciones pueden responder a requerimientos como:

- agregar características opcionales,
- incorporar servicios adicionales,
- modificar el comportamiento observable sin cambiar la identidad del objeto,
- combinar funcionalidades de forma flexible según el contexto.

Un enfoque inicial suele consistir en utilizar herencia para representar cada nueva combinación de funcionalidades. Sin embargo, cuando las variaciones comienzan a combinarse entre sí, este enfoque conduce rápidamente a un diseño rígido y difícil de mantener.

Desde una perspectiva metodológica, el problema puede formularse así:

¿Cómo extender el comportamiento de un objeto de forma flexible y combinable, sin crear una explosión de subclases ni modificar el código existente?

El patrón Decorator surge como una solución estructurada a este problema, permitiendo agregar responsabilidades a un objeto de manera dinámica, preservando su interfaz y sin recurrir a jerarquías complejas de herencia.

Ejemplo ilustrativo sin código: "El mismo objeto, distintas capas de funcionalidad"

Considérese un sistema de ventas de productos. Un producto base tiene un precio y una descripción. Sin embargo, dependiendo del contexto, el producto puede incorporar funcionalidades adicionales:

- envoltura especial,
- garantía extendida,
- envío prioritario,
- seguro adicional.

Cada una de estas características:

- no define un nuevo tipo de producto,
- puede combinarse con otras,
- puede agregarse o quitarse según la elección del cliente.

Si se modelara cada combinación mediante herencia, el sistema debería incluir clases como:

- ProductoConGarantia,
- ProductoConEnvio,
- ProductoConGarantiaYEnvio,
- ProductoConGarantiaEnvioYSeguro, etc.

Este enfoque no escala. El número de clases crece exponencialmente y el diseño pierde claridad.

Aquí aparece una señal clara para aplicar Decorator:

- cuando las funcionalidades son opcionales,
- cuando deben combinarse dinámicamente,
- cuando no representan una identidad distinta, sino una extensión del mismo objeto.

Diseño ingenuo y sus problemas estructurales

Un diseño ingenuo suele utilizar herencia para representar cada variación:

```
class Producto {
    public double getPrecio() { return 100; }
}
class ProductoConGarantia extends Producto {
    public double getPrecio() { return super.getPrecio() + 20; }
}
class ProductoConEnvio extends Producto {
    public double getPrecio() { return super.getPrecio() + 10; }
}
```

Este enfoque presenta problemas graves:

- Explosión de clases al combinar funcionalidades.
- Rigidez: agregar una nueva funcionalidad implica crear múltiples nuevas subclases.
- Dificultad de mantenimiento: cambios pequeños afectan muchas clases.
- Violación del principio Abierto/Cerrado: el sistema se modifica constantemente.

Desde el punto de vista del diseño, este es un caso típico donde la herencia deja de ser una herramienta adecuada.

Idea central del patrón Decorator

El patrón **Decorator** se basa en una idea fundamental: En lugar de extender un objeto mediante herencia, envuélvalo con otro objeto que agregue funcionalidad.

Esto implica que:

- el objeto decorado y el decorador comparten la misma interfaz,
- el decorador delega el comportamiento base al objeto envuelto,
- el decorador añade comportamiento adicional antes o después de la delegación.

De esta manera, las funcionalidades se agregan como capas, no como ramas de una jerarquía.

Formalización del diseño: pasos metodológicos

Paso 1: Definir el componente base

Se define una interfaz o clase abstracta que represente al objeto base:

```
interface Producto {
    double getPrecio();
    String getDescripcion();
}
```

Esta interfaz actúa como contrato común para el objeto base y los decoradores.

Paso 2: Implementar el componente concreto

```
class ProductoBasico implements Producto {

    @Override
    public double getPrecio() {
        return 100;
    }

    @Override
    public String getDescripcion() {
        return "Producto básico";
    }
}
```

Este es el objeto que puede ser decorado dinámicamente.

Paso 3: Definir el decorador abstracto

```
abstract class ProductoDecorator implements Producto {

    protected Producto producto;

    public ProductoDecorator(Producto producto) {
        this.producto = producto;
    }
```

```
    @Override
    public double getPrecio() {
        return producto.getPrecio();
    }

    @Override
    public String getDescripcion() {
        return producto.getDescripcion();
    }
}
```

Este paso es clave: el decorador implementa la misma interfaz y mantiene una referencia al componente.

Paso 4: Implementar decoradores concretos

```
class GarantiaExtendida extends ProductoDecorator {

    public GarantiaExtendida(Producto producto) {
        super(producto);
    }

    @Override
    public double getPrecio() {
        return super.getPrecio() + 20;
    }
```

```
    @Override
    public String getDescripcion() {
        return super.getDescripcion() + " + garantía extendida";
    }
}

class EnvioPrioritario extends ProductoDecorator {

    public EnvioPrioritario(Producto producto) {
        super(producto);
    }

    @Override
    public double getPrecio() {
        return super.getPrecio() + 10;
    }

    @Override
    public String getDescripcion() {
        return super.getDescripcion() + " + envío prioritario";
    }
}
```

Cada decorador encapsula una responsabilidad adicional, manteniendo alta cohesión.

Paso 5: Uso dinámico del patrón

```
public class Main {
    public static void main(String[] args) {

        Producto producto = new ProductoBasico();
        producto = new GarantiaExtendida(producto);
        producto = new EnvioPrioritario(producto);

        System.out.println(producto.getDescripcion());
        System.out.println("Precio total: " + producto.getPrecio());
    }
}
```

Las funcionalidades se agregan en tiempo de ejecución, sin modificar clases existentes.

Valor metodológico

Decorator enseña al estudiante que:

- no toda extensión debe resolverse con herencia,
- las responsabilidades pueden componerse dinámicamente,
- el diseño puede crecer sin romper jerarquías existentes,
- la composición es una herramienta poderosa frente a la rigidez.

Decorator complementa a Strategy y Observer al abordar cómo extender funcionalidades sin modificar

clases existentes. Mientras Strategy controla variaciones de comportamiento y Observer controla reacciones a eventos, Decorator controla la agregación progresiva de responsabilidades.

En la siguiente sección se estudiará el patrón Singleton, analizando no solo su implementación, sino también sus riesgos y criterios de uso, para que el estudiante comprenda cuándo es adecuado y cuándo debe evitarse.

4.6. El patrón Singleton: una única instancia con acceso controlado

En sistemas orientados a objetos, existen componentes que, por su naturaleza, no deberían multiplicarse dentro de la ejecución del programa. No porque "sea más cómodo", sino porque representan un recurso único o un punto de coordinación central. Por ejemplo:

- un gestor de configuración (valores del sistema, rutas, credenciales),
- un registro central de auditoría,
- un gestor de conexión a una cola de mensajes,
- un controlador de acceso a un archivo crítico,
- un gestor de cache compartida.

Cuando este tipo de componente se instancia más de una vez, pueden aparecer problemas reales:

- configuraciones inconsistentes (cada módulo lee una configuración diferente),
- duplicación de recursos (múltiples conexiones innecesarias),
- registros de auditoría fragmentados,
- colisiones en escritura de archivos,

- estados imposibles de depurar porque “no existe un único punto de verdad”.

En este contexto, el patrón Singleton propone una estructura para garantizar que una clase tenga una única instancia durante la ejecución y que exista un punto de acceso controlado a esa instancia.

Sin embargo, a nivel universitario es importante enseñar Singleton de manera crítica: es un patrón útil, pero también uno de los más mal usados. En consecuencia, este apartado no solo muestra “cómo implementarlo”, sino cuándo conviene y cuándo no, y qué señales indican un uso incorrecto.

Ejemplo ilustrativo sin código: “Configuración única del sistema”

Imagine un sistema académico con varios módulos: matrícula, seguimiento a graduados, reportes, notificaciones y auditoría. Todos necesitan conocer parámetros como:

- nombre de la institución,
- ruta del repositorio de reportes,
- modo de ejecución (producción/pruebas),
- claves o endpoints de servicios,
- reglas de formato de reportes (fechas, moneda, zona horaria).

En un diseño ingenuo, cada módulo podría “crear su propia configuración” o cargarla desde archivo cuando lo necesite. El sistema aparentemente funciona, pero la realidad es que ese diseño abre varias puertas a fallos:

- un módulo se ejecuta con parámetros viejos porque cargó antes el archivo,
- otro módulo lee otra ruta,

- los cambios se vuelven inconsistentes,
- se repite código de carga de configuración en múltiples lugares,
- aparecen “bugs fantasma” difíciles de reproducir.

En esta situación, el sistema necesita una regla clara:

Debe existir una única configuración activa, compartida por todo el sistema.

Aquí aparece la necesidad de Singleton, porque el dominio exige una única instancia coherente y accesible.

Los bugs fantasmas (también llamados heisenbugs en la literatura técnica) son errores de software que aparecen y desaparecen sin un patrón claro, son difíciles de reproducir y, muchas veces, parecen “desvanecerse” cuando se intenta analizarlos o depurarlos.

En términos académicos y prácticos, no son errores imaginarios, sino fallos reales cuyo origen está en un diseño deficiente, estados inconsistentes o dependencias ocultas.

Diseño ingenuo y sus problemas estructurales

Un diseño ingenuo típico luce así (idea conceptual):

- Cada clase crea su propio objeto Configuracion.
- Cada objeto Configuracion carga el archivo o define valores.
- Como resultado, puede haber múltiples configuraciones en memoria.

Los problemas estructurales son:

1. Inconsistencia de estado: Si hay más de una instancia, es posible que contengan valores distintos.
2. Duplicación de trabajo y recursos: Se carga el archivo varias veces o se consulta varias veces una fuente externa.
3. Dificultad de mantenimiento: La lógica de "cargar configuración" queda repetida en varias clases.
4. Diagnóstico complejo: Cuando el sistema falla, no es evidente qué módulo está operando con qué configuración.

En términos metodológicos, el sistema pierde un principio esencial: un único punto de verdad para parámetros globales.

Idea central del patrón Singleton

El patrón Singleton formaliza una solución con tres condiciones:

1. Constructor privado: nadie puede crear instancias desde fuera.
2. Campo estático: la instancia única se mantiene en la propia clase.
3. Método público de acceso: un método controla el acceso y devuelve siempre la misma instancia.

Esto no significa que Singleton sea "un objeto global sin control". La idea correcta es:

Un Singleton es una instancia única controlada, no una variable global arbitraria.

Formalización del diseño (paso a paso aplicado al ejemplo de Configuración)

En este libro se presenta la implementación en Java en una progresión pedagógica: cada paso responde a una necesidad y se justifica.

Paso 1: Definir una clase que represente la configuración

El primer paso es modelar la clase con los datos que se consultarán frecuentemente.

```
class ConfiguracionSistema {
    private String institucion;
    private String rutaReportes;
    private String modoEjecucion;

    // getters
}
```

El estudiante debe comprender que Singleton no es "un patrón mágico", sino una decisión de diseño aplicada a una clase que representa un recurso único.

Paso 2: Evitar instanciación externa (constructor privado)

El constructor se hace privado para impedir new ConfiguracionSistema() desde cualquier parte del sistema.

```
public class ConfiguracionSistema {

    private ConfiguracionSistema() {
        // cargar configuración
    }
```

```
}
```

Evidencia del paso:

- Si el constructor es privado, el compilador impide instanciar desde fuera.
- Se fuerza a que el acceso pase por un único mecanismo.

Por qué era necesario: Sin constructor privado, no hay garantía real de unicidad.

Paso 3: Crear el atributo estático que almacenará la instancia única

```
private static ConfiguracionSistema instancia;
```

Aquí aparece la noción de "única instancia para toda la JVM (Java Virtual Machine, por sus siglas en inglés)".

El atributo estático pertenece a la clase, no a un objeto. Por tanto, no se duplica por instanciación.

Paso 4: Crear el método público de acceso (control de creación)

```
public static ConfiguracionSistema getInstancia() {
    if (instancia == null) {
        instancia = new ConfiguracionSistema();
    }
    return instancia;
}
```

Esta versión se conoce como lazy initialization: se crea la instancia solo cuando se necesita.

Evidencia del paso:

- La primera vez se crea.

- Las siguientes veces se reutiliza.

Ventaja directa en el ejemplo:

- El archivo de configuración se carga una sola vez.
- Toda la aplicación comparte los mismos valores.

Implementación completa


```
public class ConfiguracionSistema {

    private static ConfiguracionSistema instancia;

    private String institucion;
    private String rutaReportes;
    private String modoEjecucion;

    private ConfiguracionSistema() {
        // Simulación de carga (en un caso real se leería de
archivo/BD/env)
        this.institucion = "UMET";
        this.rutaReportes = "C:/reportes/";
        this.modoEjecucion = "PRODUCCION";
    }

    public static ConfiguracionSistema getInstancia() {
        if (instancia == null) {
            instancia = new ConfiguracionSistema();
```

```
        }
        return instancia;
    }

    public String getInstitucion() { return institucion; }
    public String getRutaReportes() { return rutaReportes; }
    public String getModoEjecucion() { return modoEjecucion;
}
}
```

Demostración de uso (main) con interpretación pedagógica

```
public class MainConfiguracion {
    public static void main(String[] args) {

        ConfiguracionSistema c1 = ConfiguracionSistema.
getInstancia();
        ConfiguracionSistema c2 = ConfiguracionSistema.
getInstancia();

        System.out.println("Institución: " + c1.getInstitucion());
        System.out.println("Ruta reportes: " + c2.getRutaRe-
portes());

        System.out.println("¿Misma instancia?: " + (c1 == c2));
    }
}
```

En el ejemplo de Configuración del Sistema, el patrón Singleton no se utiliza por "tradición" ni por moda: se utiliza porque existe una necesidad real de diseño: garantizar un único estado coherente compartido por todos los módulos.

Observe cómo cada parte del patrón responde a un riesgo concreto:

- El constructor privado evita que cada módulo cree su propia configuración, eliminando el problema de configuraciones duplicadas o contradictorias.
- El campo estático instancia actúa como el único punto en memoria donde la configuración existe, y por tanto se convierte en una referencia estable para todo el sistema.
- El método getInstancia() funciona como un controlador de acceso, asegurando que la configuración se cree una sola vez y luego se reutilice, lo cual reduce costos de carga y evita inconsistencias.

Desde una perspectiva pedagógica, el aprendizaje clave es este:

Singleton no es "una clase especial", sino una solución a un problema concreto: cuando el dominio requiere un único punto de verdad, el diseño debe impedir la multiplicación accidental de instancias.

Además, el main demuestra una evidencia verificable para el estudiante: al comparar c1 == c2 se confirma que ambas referencias apuntan al mismo objeto. Esto no es un detalle menor: es la prueba práctica de que el sistema está compartiendo la misma configuración y que, por tanto, cualquier módulo consultará exactamente los mismos valores.

En síntesis, el patrón aporta al diseño del sistema:

- Consistencia global (misma configuración para todos),
- Ahorro de recursos (carga única),
- Simplificación del acceso (un punto de acceso controlado),
- Mejor mantenibilidad (centralización clara de parámetros).

Advertencias: cuándo NO conviene usar Singleton

A nivel formativo, es imprescindible aclarar que Singleton puede ser mal aplicado. No se recomienda cuando:

1. **Se usa como "variable global" para todo:** Si una clase "Singleton" empieza a almacenar de todo, se convierte en una clase Dios.
2. **Se necesita testear con facilidad:** Los Singletons pueden dificultar pruebas unitarias si se usan como dependencia rígida.
3. **Existen múltiples configuraciones por contexto:** Por ejemplo: multi-tenant, múltiples sedes o entornos simultáneos.
4. **Se usa para ocultar mal diseño:** Si el motivo real es "no sé cómo pasar dependencias", Singleton se vuelve un parche.

Regla metodológica:

Singleton debe representar un recurso verdaderamente único. Si la unicidad no es una necesidad del dominio, debe evitarse.

En la siguiente sección se realizará un **cierre integrador** del capítulo, comparando los cinco patrones en términos de "qué problema resuelven", "qué error evitan" y "qué tipo de variación controlan", consolidando el criterio del estudiante para seleccionar el patrón adecuado.

4.7. Selección razonada de patrones: comparación explicativa y caso integrador

En los epígrafes 4.1 a 4.5 se estudiaron cinco patrones fundamentales, cada uno asociado a un tipo de problema distinto: variación de comportamiento (Strategy), creación desacoplada (Factory Method), reacción a cambios (Observer), extensión dinámica de funcionalidades (Decorator) y unicidad de instancia (Singleton). El siguiente paso no es "memorizar" patrones, sino aprender a decidir con criterio cuál patrón aplica según el problema real y por qué los otros no son la mejor opción en ese caso. Por ello, esta sección funciona como un mapa de decisión: integra comparación, señales del problema y razonamiento, y culmina con un caso integrador breve que obliga a justificar elecciones.

Tabla: qué problema resuelve cada patrón y cómo reconocerlo. Cómo usar esta tabla

El estudiante debe leerla como un diagnóstico: primero identifica el síntoma del diseño (qué duele), luego observa qué patrón controla esa variación y finalmente comprende por qué otras soluciones suelen ser incorrectas (Tabla 4.1).

Tabla 4.1. Problema que resuelve cada patrón y cómo reconocerlo.

Patrón	¿Qué "variación" controla?	Señales claras de que lo necesitas (síntomas del diseño)	Decisión correcta (qué haces)	Por qué NO usar otro patrón en su lugar (errores típicos)	Pregunta guía (diagnóstico rápido)
Strategy	Variación de comportamiento (algoritmos o reglas que cambian)	Muchos if/switch para elegir "cómo calcular", "cómo validar", "cómo cobrar". Se agregan nuevos casos y el método central crece sin control. La regla cambia frecuentemente por negocio o contexto.	Extraes el comportamiento variable a clases intercambiables con una interfaz común. El "contexto" queda estable y delega.	No Factory: Factory crea objetos, no encapsula reglas de cálculo. No Decorator: Agrega responsabilidades, no sustituye algoritmos. No Singleton: Un único objeto no resuelve variación de reglas.	¿Lo que cambia es el "cómo se hace" una operación?
Factory Method	Variación de creación (qué clase concreta instanciar)	Código cliente lleno de new ClaseConcreta(). Un lugar central decide "qué crear" con if/switch. Aparecen nuevos tipos y hay que modificar código existente para crearlos.	Centralizas el proceso estable y delegas en subclases la decisión de qué producto concreto instanciar.	No Strategy: Strategy cambia comportamiento en runtime, no formaliza creación por familias/tipos. No Singleton: Singleton controla unicidad, no selección de tipo. No Observer: Observer notifica cambios, no construye productos.	¿El problema es "qué objeto debo crear" según un tipo o contexto?

Observer	Variación de reacciones a un evento (quién debe enterarse)	Cuando algo cambia, hay que avisar a muchos módulos. El objeto "principal" conoce a todos los demás (alto acoplamiento). Agregar una nueva reacción obliga a modificar al emisor.	El emisor (sujeto) notifica; los observadores se suscriben y reaccionan sin que el sujeto conozca detalles.	No Strategy: Strategy elige un algoritmo, no gestiona suscriptores. No Decorator: Decorator no "difunde eventos". No Factory: Factory no resuelve coordinación de módulos.	¿El problema es "cuando pasa X, varios deben reaccionar"?
Decorator	Variación por combinación de funcionalidades (capas opcionales)	Funcionalidades opcionales que se combinan (A + B + C). Herencia produce explosión de clases ("ConXConYConZ"). Se quiere agregar/quitar responsabilidades dinámicamente.	Envuelves un objeto con decoradores que añaden responsabilidad manteniendo la misma interfaz.	No Strategy: Strategy sustituye algoritmos; Decorator agrega capas sin sustituir identidad base. No Factory: Factory crea; Decorator extiende objetos ya creados. No Singleton: Unicidad no ayuda a combinar capas.	¿El problema es "quiero agregar/quitar características combinables"?

Sin-gleton	Variación/ control de unicidad (un solo punto de verdad)	Varios módulos requieren compartir el mismo estado global legítimo (configuración, cache, logger). Múltiples instancias generan inconsistencias o recursos duplicados. Se necesita un punto de acceso controlado.	Restringes instanciación (constructor privado) y expones un único acceso global controlado.	No Factory: Factory crea familias de objetos; Singleton restringe a uno. No Observer: Observer no controla instancias. No Decorator/Strategy: no controlan unicidad.	¿Debe existir una sola instancia compartida en toda la ejecución?

Guía metodológica de decisión: pasos para elegir patrón sin adivinar

Para evitar "elegir patrones por moda", se propone este procedimiento de análisis (que el estudiante debe practicar):

1. Nombrar el punto de cambio: ¿Qué cambia en el sistema: reglas, tipos, reacciones, funcionalidades, unicidad?
2. Clasificar la variación

- Cambia el cómo se hace: Strategy
- Cambia el qué se crea: Factory Method
- Cambia el quién reacciona: Observer

- Cambian capas combinables: Decorator
- Cambia el riesgo de múltiples instancias: Singleton

3. Identificar el síntoma dominante (lo que hoy rompe el diseño): El patrón se elige por el dolor principal: condicionales, acoplamiento, explosión de clases, etc.
4. Justificar por descarte

Una buena decisión debe explicar:

- por qué el patrón seleccionado sí resuelve el síntoma,
- por qué los otros no atacan el núcleo del problema.

Caso integrador: "Clínica VitalCare: servicios, planes y notificaciones"

Descripción del dominio

Una clínica privada "VitalCare" gestiona:

- Planes de atención (Básico, Familiar, Premium).
- Cálculo de copago según plan y tipo de servicio.
- Notificaciones automáticas a módulos internos cuando se registra una cita (ej.: auditoría, mensajería, estadísticas).
- Servicios adicionales opcionales al momento de reservar (recordatorio por SMS, prioridad en agenda, servicio a domicilio).
- Configuración única (tarifas base, impuestos, rutas de reportes) compartida por toda la aplicación.

La dirección informa además que:

- Las reglas de copago cambian cada semestre (por políticas).

- Los servicios adicionales son combinables (un paciente puede elegir varios).
- Se agregarán nuevos planes en el futuro.
- Cada vez que se registra una cita, se deben disparar acciones en varios módulos sin acoplarlos.

Diagnóstico del problema (identificación de variaciones)

Aquí no existe una sola variación: aparecen al menos cinco tipos de cambio, cada uno asociado a un patrón distinto:

1. Variación de comportamiento: cálculo de copago cambia con frecuencia.
2. Variación de creación: nuevos planes futuros (crear instancias de Plan).
3. Variación de reacciones: varios módulos reaccionan al registrar cita.
4. Variación por capas combinables: servicios adicionales opcionales.
5. Control de unicidad: configuración global única.

El valor pedagógico del caso es mostrar que los patrones no compiten, sino que se complementan cuando el dominio lo exige.

Decisión razonada patrón por patrón (con "por qué sí" y "por qué no")

A) Strategy para el cálculo del copago

Decisión correcta: Strategy.

Por qué sí Strategy

- La regla de copago es un algoritmo que cambia con frecuencia (semestre a semestre).

- Si se codifica dentro de una clase central (ej. CalculadoraCopago con if/switch), se repite el error del Capítulo 3: explosión de condicionales.
- Strategy permite encapsular cada regla de copago en clases independientes, dejando estable el contexto.

Por qué no solo herencia

- Hacer una jerarquía de "PlanBasicoConCopagoX" no escala: el copago cambia y obligaría a modificar o multiplicar clases.

Por qué no Decorator

- Decorator sirve para agregar responsabilidades combinables (capas), no para sustituir reglas de cálculo centrales que deben cambiar por política.

B) Factory Method para crear planes sin acoplar al cliente

Decisión correcta: Factory Method (o familia de fábricas de planes).

Por qué sí Factory Method

- El sistema debe crear un tipo concreto de Plan según el contexto (Básico/Familiar/Premium), y crecer en el futuro sin modificar al cliente.
- Factory Method evita que el resto del sistema dependa de new PlanPremium() dispersos.

Por qué no Strategy

- Strategy controla "cómo calcular", no "qué clase instanciar". Un plan no es un algoritmo intercambiable: es un objeto con identidad y estructura.

Por qué no Singleton

- No puede haber un único plan; debe existir uno por paciente o por contrato. Singleton rompería el dominio.

C) Observer para notificar módulos cuando se registra una cita

Decisión correcta: Observer.

Por qué sí Observer

- Registrar una cita es un evento; hay múltiples reacciones: auditoría, mensajería, estadísticas.
- El sistema debe permitir agregar nuevas reacciones sin modificar la clase que registra la cita.

Por qué no Strategy

- Strategy elige un algoritmo; aquí no se elige un algoritmo: se notifica a múltiples dependientes.

Por qué no Factory Method

- Factory crea objetos, pero no gestiona "quién reacciona" ante cambios.

D) Decorator para servicios adicionales combinables

Decisión correcta: Decorator.

Por qué sí Decorator

- Los servicios extra (SMS, prioridad, domicilio) son opcionales y combinables.
- Decorator permite envolver una "ReservaBase" con capas adicionales, sumando costo y descripción sin explosión de clases.

Por qué no herencia

- Herencia produciría combinaciones:

  ReservaConSmsConPrioridadConDomicilio, etc.

Por qué no Strategy

- No se está sustituyendo un algoritmo único, se están agregando capas acumulativas.

E) Singleton para configuración única de la clínica

Decisión correcta: Singleton (con criterio y uso responsable).

Por qué sí Singleton

- Tarifa base, impuestos y rutas deben ser coherentes globalmente.
- Múltiples instancias de configuración pueden generar inconsistencias y bugs difíciles de reproducir.

Por qué no Factory Method

- Factory crea múltiples productos. Configuración debe ser única.

Advertencia metodológica

- Singleton no debe usarse como "bolsa global de datos"; debe representar un recurso realmente único.

Este caso demuestra que un sistema profesional no se diseña eligiendo "un patrón favorito", sino identificando qué tipo de variación existe y asignándole una solución estructurada. En la clínica VitalCare, el error típico del diseño ingenuo sería concentrar todo en una clase central (por ejemplo, SistemaClinica) con condicionales, creación directa con new, notificaciones manuales y combinaciones por

herencia. Ese enfoque funcionaría solo al inicio, pero se volvería frágil cuando:

- cambien las reglas de copago,
- se agreguen nuevos planes,
- aparezcan nuevos módulos que deban reaccionar,
- se combinen múltiples servicios adicionales,
- la configuración cambie o se cargue en distintos momentos.

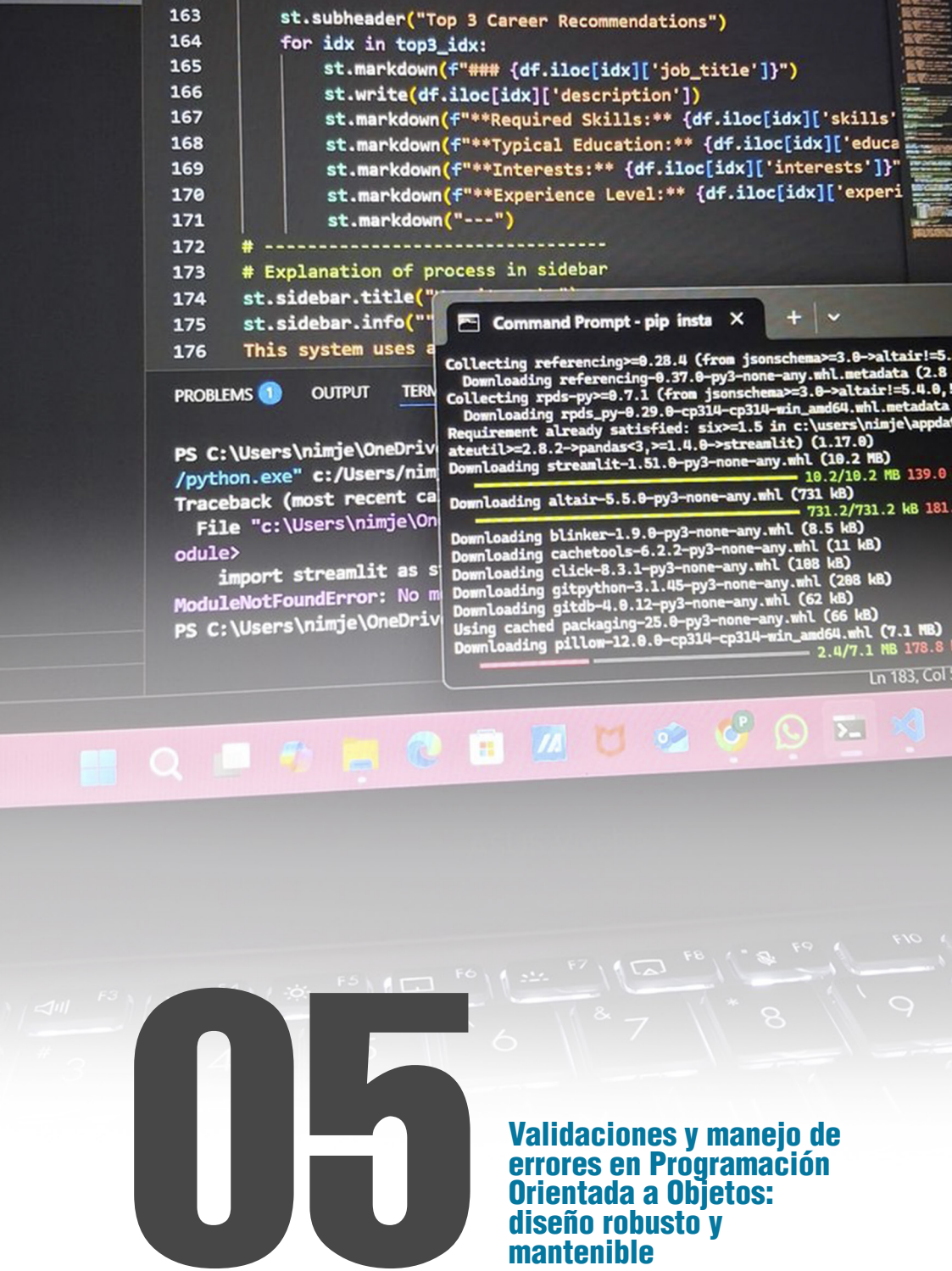
La combinación de Strategy, Factory Method, Observer, Decorator y Singleton permite que cada variación esté controlada en el lugar correcto:

- Strategy protege el cálculo frente a cambios frecuentes,
- Factory protege la creación frente a nuevos tipos,
- Observer protege la coordinación frente a nuevos módulos,
- Decorator protege la extensión frente a combinaciones,
- Singleton protege la coherencia frente a multiplicidad de estado global.

El estudiante debe comprender que la "respuesta correcta" no es el patrón en sí, sino la justificación basada en el dominio y en los síntomas del diseño. Acorde a lo expresado por Fernández-Marín et al. (2024) donde expresa que la justificación explícita de decisiones de diseño resulta fundamental en contextos académicos y de investigación, donde el software no solo debe funcionar, sino ser comprensible, extensible y evaluable como artefacto científico

En este capítulo se demostró que los patrones de diseño no son “trucos” de programación, sino decisiones estructurales que permiten controlar el cambio sin degradar la calidad del sistema. Strategy, Factory Method, Observer, Decorator y Singleton aportan criterios para evitar condicionales explosivos, creación acoplada, notificaciones rígidas, jerarquías forzadas y estados inconsistentes, reforzando cohesión, reduciendo acoplamiento y mejorando la mantenibilidad.

Sin embargo, un sistema bien diseñado no se construye solo con patrones: también exige disciplina en aspectos cotidianos que, si se descuidan, rompen la calidad incluso en diseños aparentemente correctos. Entre ellos destacan la validación de entradas, el manejo coherente de errores, la definición de reglas de negocio explícitas y la comunicación clara de fallos hacia capas superiores. Por ello, el siguiente capítulo retoma estos elementos “de programación diaria” y los integra con lo aprendido sobre diseño de clases, responsabilidades y patrones, para que el estudiante consolide una forma profesional de construir software robusto.



05

Validaciones y manejo de errores en Programación Orientada a Objetos: diseño robusto y mantenible

5.1. Tipos de errores: clasificación que todo programador debe dominar

En el desarrollo de software orientado a objetos, uno de los aspectos más determinantes para la calidad de un sistema es la forma en que se gestionan las situaciones inválidas, las reglas de negocio y los errores inevitables. La correcta gestión de errores y validaciones constituye un componente esencial de la calidad del software educativo e institucional, especialmente en sistemas que apoyan procesos críticos de toma de decisiones y análisis de información (Fernández Marín et al., 2022). A pesar de su importancia, estos temas suelen abordarse de manera superficial durante la formación académica, limitándose al uso mecánico de estructuras como if o bloques try-catch. Esta aproximación resulta insuficiente cuando los sistemas crecen en complejidad y comienzan a interactuar múltiples módulos, capas y actores. En estos escenarios, una validación mal ubicada o un error mal gestionado puede comprometer la estabilidad, la mantenibilidad y la claridad del software. Por ello, comprender cómo diseñar correctamente las validaciones y el manejo de errores es un paso indispensable para evolucionar desde una programación básica hacia una práctica profesional. Este capítulo se enfoca precisamente en cubrir ese vacío formativo.

A lo largo de los capítulos anteriores se han estudiado los principios fundamentales de la Programación Orientada a Objetos, el diseño correcto de clases, el control del crecimiento del sistema y la aplicación de patrones de diseño básicos. Sin embargo, incluso un diseño estructuralmente correcto puede degradarse rápidamente si no se definen reglas claras sobre cómo validar datos, cómo expresar violaciones del dominio y cómo comunicar errores entre las distintas partes del sistema. En la práctica profesional, muchos

problemas no se originan en algoritmos complejos, sino en entradas inválidas, supuestos incorrectos o excepciones mal utilizadas. Cuando estas situaciones no se gestionan de forma coherente, el código tiende a llenarse de condicionales repetidos, mensajes dispersos y dependencias implícitas. Este capítulo retoma los conceptos ya aprendidos y los integra en un enfoque sistemático para el manejo de estos escenarios. De esta manera, se refuerza la idea de que la robustez del software es una consecuencia directa de buenas decisiones de diseño.

El manejo de errores y las validaciones no deben entenderse como tareas accesorias ni como simples comprobaciones técnicas, sino como parte integral del modelo del dominio y de la lógica del sistema. Validar correctamente implica decidir dónde debe vivir una regla, quién es responsable de hacerla cumplir y cómo debe reaccionar el sistema cuando dicha regla se viola. De igual forma, lanzar o capturar una excepción no es una acción trivial, sino una decisión que afecta el flujo de ejecución, la legibilidad del código y la experiencia del usuario o del desarrollador que consume ese componente. En este capítulo se mostrará cómo separar validaciones de casos de uso, cómo diseñar excepciones de negocio significativas y cómo evitar el uso indiscriminado de estructuras de control que deterioran el diseño. Cada concepto será desarrollado de forma progresiva, apoyándose en ejemplos claros y casos de estudio resueltos. El objetivo es que el estudiante no solo entienda qué hacer, sino también por qué hacerlo de esa manera.

La formación en programación y diseño de software debe concebirse como un proceso orientado al desarrollo de competencias digitales, donde la comprensión de algoritmos, estructuras de código y mecanismos de control de errores resulta tan relevante

como su correcta implementación técnica. En este sentido, propuestas educativas de carácter interactivo evidencian que el análisis reflexivo del software, incluyendo la identificación de errores, decisiones de diseño y evaluación de soluciones, contribuye significativamente al fortalecimiento de dichas competencias en contextos universitarios (Fernández Marín et al., 2022).

Finalmente, este capítulo busca consolidar una visión profesional de la programación orientada a objetos, en la que la corrección funcional del sistema no es suficiente si no va acompañada de claridad, coherencia y capacidad de evolución. Aprender a validar y manejar errores de forma estructurada permite construir sistemas más confiables, más fáciles de probar y menos propensos a fallos difíciles de diagnosticar. Además, estas prácticas preparan al estudiante para enfrentar escenarios reales de desarrollo, donde los errores son inevitables y deben ser gestionados con criterio y responsabilidad. A lo largo del capítulo se integrarán principios de responsabilidad única, bajo acoplamiento y patrones de diseño, mostrando su aplicación concreta en problemas cotidianos de programación. Con ello, se pretende que el lector desarrolle una forma de pensar el software que trascienda el código inmediato y se enfoque en la calidad global del sistema. Este enfoque servirá como base para los capítulos siguientes, donde se abordará la organización estructural del software a mayor escala.

¿Por qué este capítulo es crítico en la formación del programador?

En la práctica profesional, la mayoría de los errores de software no provienen de algoritmos complejos, sino de:

- datos inválidos,

- reglas de negocio mal aplicadas,
- errores mal comunicados,
- excepciones mal usadas o ignoradas.

Paradójicamente, estos temas suelen tratarse de forma superficial en cursos básicos, reduciéndose a "usar try-catch" sin criterio. Este capítulo corrige esa carencia, mostrando que validar y manejar errores es una decisión de diseño, no un detalle técnico.

Uno de los errores más frecuentes en estudiantes, e incluso en programadores en ejercicio, es tratar todos los errores de la misma manera. En muchos casos, cualquier situación inválida se resuelve con un if, un try-catch genérico o, peor aún, con un mensaje impreso por pantalla. Este enfoque improvisado genera sistemas confusos, difíciles de mantener y con comportamientos inconsistentes.

Antes de escribir una sola línea de código, el programador debe detenerse a analizar qué tipo de error está enfrentando. No todos los errores tienen el mismo origen, no todos tienen la misma gravedad y, por lo tanto, no todos deben resolverse de la misma forma. Enseñar esta clasificación desde etapas tempranas permite al estudiante desarrollar pensamiento crítico y tomar decisiones de diseño más acertadas. Desde una perspectiva didáctica, esta sección busca que el estudiante aprenda a identificar, clasificar y reaccionar ante distintos tipos de errores, entendiendo que una mala clasificación conduce a malas soluciones, mientras que una buena clasificación simplifica el diseño y mejora la claridad del sistema.

Los errores de entrada son aquellos que se producen cuando el usuario o una interfaz externa proporciona datos inválidos. Estos errores no indican que el sistema esté fallando, sino que el sistema recibió información

incorrecta o incompleta. Ejemplos típicos de este tipo de error incluyen:

- campos vacíos en un formulario,
- formatos incorrectos (correo sin @, fecha mal escrita),
- valores fuera de rango (edad negativa, cantidad cero).
- escribe letras donde van números,
- copia y pega valores incorrectos,

Estos errores son esperables. En un sistema real, el usuario se equivoca, escribe mal o deja campos incompletos, y el software debe estar preparado para ello. Por esta razón, no deben tratarse como situaciones excepcionales, sino como parte normal del flujo del sistema. Por eso lo correcto es validar, informar claramente y pedir corrección sin lanzar excepciones "como si el sistema colapsara".

Ejemplo 1: Campo vacío (nombre)

Situación: Un sistema de registro de estudiantes solicita "nombre".

- El usuario lo deja vacío.
- Entrada recibida: "" (cadena vacía)
- Problema: no hay información para registrar a nadie

Esto es un error de entrada, porque el dato no cumple el requisito mínimo: "ser un texto no vacío".

Mala solución (rompe el flujo como si fuera un fallo grave)

```
if (nombre.equals("")) {
    throw new RuntimeException("Error");
}
```

Por qué es mala:

- Usa una excepción genérica (no explica nada).
- Trata el error del usuario como si fuera un fallo del sistema.
- No guía al usuario (solo "explota").

Buena solución (validación más mensaje claro)

```
if (nombre == null || nombre.trim().isEmpty()) {
    return ResultadoValidacion.error("El nombre es obligatorio y no puede estar vacío". );
}
```

Por qué es buena:

- El sistema sigue estable.
- El usuario sabe qué corregir.
- Se controla el flujo.

Ejemplo 2: Formato incorrecto (correo)

Situación: El usuario escribe un correo sin @.

Entrada: "alexis.umet.edu.ec"

Problema: no cumple formato básico de correo.

Esto no es "regla de negocio". No es "error técnico". Es un error de entrada porque el dato está mal formado.

Mala solución (acepta el dato y "deja que falle después")

```
// se guarda tal como viene
usuario.setCorreo(correo);
```

Por qué es mala:

- El error se esconde y aparece después en otro módulo.

- Se generan “bugs fantasma” (aparece un fallo lejos de la causa).
- El sistema queda con datos sucios.

Buena solución (validar antes)

```
if (correo == null || !correo.contains(“@”)) {
    return ResultadoValidacion.error(“Correo inválido. Debe contener ‘@’”. );
}
```

Por qué es buena:

- Corta el problema donde nace.
- Mantiene el sistema consistente.
- Evita que se almacene basura.

Ejemplo 3: Valor fuera de rango (edad negativa)

Situación: Usuario ingresa edad = -5.

Entrada: -5

Problema: una edad negativa no tiene sentido.

Este caso no depende del negocio, porque en cualquier dominio una edad negativa es inválida. Es un error de entrada, por rango.

Mala solución (parche improvisado)

```
if (edad < 0) edad = 0; // “arreglo” automático
```

Por qué es mala:

- Esconde el error.
- Modifica datos sin permiso.
- Se vuelve imposible saber qué pasó realmente.

Buena solución (validación explícita)

```
if (edad < 0 || edad > 120) {

    return ResultadoValidacion.error("Edad inválida. Debe
estar entre 0 y 120". );

}
```

Por qué es buena:

- La regla es explícita.
- No altera datos sin control.
- El usuario puede corregir.

Clase centralizada de validación para errores de entrada: ¿por qué es importante y qué función cumple?

Una decisión de diseño que, aunque parece pequeña, marca una diferencia enorme en la calidad del sistema es centralizar la validación de entradas en una clase (o conjunto de clases) dedicada a esa responsabilidad. En la práctica, muchos sistemas estudiantiles fallan no por algoritmos complejos, sino porque las validaciones aparecen "regadas" por todo el código: un if en el formulario, otro en el servicio, otro en el repositorio. Ese estilo disperso provoca tres problemas clásicos: duplicación, inconsistencia y mantenimiento costoso. Cuando las reglas de entrada cambian (por ejemplo, se decide que el teléfono debe admitir 9 dígitos o que el RUC debe validar prefijos), el equipo termina corrigiendo en múltiples lugares, y casi siempre se olvida uno, generando fallos difíciles de detectar.

Centralizar validaciones no significa "hacer una clase Dios" que controle todo el sistema. Significa lo contrario: separar una responsabilidad específica (validar entradas) y concentrarla donde corresponde, para que el resto del sistema se mantenga limpio. En

un enfoque bien diseñado, la clase centralizada de validación cumple funciones claras:

- Aplicar reglas de forma consistente. La misma regla se evalúa igual en cualquier parte del sistema, evitando que un módulo acepte datos que otro rechaza.
- Evitar duplicación. Una regla se escribe una sola vez y se reutiliza, lo que reduce errores y simplifica cambios.
- Proteger el dominio. Las capas internas (servicios de negocio, modelos) reciben datos ya validados o, al menos, reciben un resultado claro de por qué no son válidos.
- Mejorar claridad y trazabilidad. Si hay un problema de entrada, se sabe dónde buscar: en la validación.
- Facilitar pruebas. Validar en una clase dedicada permite escribir pruebas unitarias simples y completas: "dado este dato, debe fallar con este mensaje".

Esta decisión encaja de forma directa con los modelos de diseño ya trabajados:

- Responsabilidad única (Capítulo 2, SRP): una clase de validación tiene una sola misión: verificar entradas. No registra usuarios, no guarda en base de datos, no envía correos. Esto evita mezclar responsabilidades.
- Evitar clases Dios (Capítulo 2): centralizar no es lo mismo que "hacerlo todo". La clave es que la clase centralizada se limite estrictamente a validación. Si empieza a hacer persistencia o reglas de negocio, entonces sí se convierte en clase Dios.

- Alta cohesión: todas las funciones dentro del validador están relacionadas: validar teléfonos, IDs, fechas, rangos. Eso es cohesión.
- Bajo acoplamiento: el servicio que registra un usuario no necesita saber cómo se valida un teléfono; solo consume un resultado (ResultadoValidacion). Si mañana cambia la regla del teléfono, el servicio no cambia.
- Control del crecimiento (Capítulo 3): sin una estrategia, los sistemas crecen a base de condicionales dispersos. La validación centralizada reduce la "explosión de condicionales" porque concentra criterios en un punto.
- Patrones (Capítulo 4): aunque aquí no se está aplicando formalmente un patrón GoF, sí se está usando una idea de diseño que se relaciona con Strategy y con diseño orientado a contratos: el servicio depende de un resultado, no de una implementación interna.

¿Por qué usar ResultadoValidacion en lugar de excepciones en errores de entrada?

En este capítulo se adoptó una postura deliberada: los errores de entrada se validan y se comunican, no se lanzan como fallas. El objeto ResultadoValidacion permite expresar un error de forma controlada, sin romper el flujo de ejecución ni obligar a envolver todo en try-catch. Además, es más pedagógico porque obliga al estudiante a pensar en el flujo: "si no es válido, no continúo", en lugar de "tiro una excepción y que alguien más lo arregle".

Ese objeto también actúa como contrato de comunicación entre capas: la validación dice si se puede continuar y explica por qué no. Más adelante, cuando se trabajen errores de negocio y errores técnicos, sí

tendrá sentido introducir excepciones especializadas; pero para entrada, este enfoque mantiene el sistema estable y la lógica más clara.

El objeto ResultadoValidacion como contrato de validación

Rol del objeto ResultadoValidacion

Para que la validación de errores de entrada sea realmente reutilizable, clara y escalable, es necesario formalizar un objeto de resultado que represente el estado de una validación. Este objeto no solo indica si los datos son válidos o no, sino que actúa como un contrato de comunicación entre capas del sistema. En lugar de lanzar excepciones o devolver valores booleanos ambiguos, el objeto ResultadoValidacion encapsula:

- el estado de la validación,
- los mensajes de error asociados,
- y la posibilidad de acumular múltiples errores en una sola operación.

Este enfoque es especialmente importante en sistemas reales, donde un formulario o una operación puede fallar por más de una razón al mismo tiempo.

Diseño básico del objeto ResultadoValidacion

```
import java.util.ArrayList;
import java.util.List;

public class ResultadoValidacion {

    private boolean valido;
    private List<String> errores;
```

```
    private ResultadoValidacion(boolean valido) {
        this.valido = valido;
        this.errores = new ArrayList<>();
    }

    public static ResultadoValidacion ok() {
        return new ResultadoValidacion(true);
    }

    public static ResultadoValidacion error(String mensaje) {
        ResultadoValidacion r = new ResultadoValidacion(false);
        r.errores.add(mensaje);
        return r;
    }

    public boolean esValido() {
        return valido;
    }

    public List<String> getErrores() {
        return errores;
    }
}
```

Soporte para acumulación de errores

Uno de los errores más comunes en implementaciones académicas es detener la validación en el primer fallo, obligando al usuario a corregir campo por campo. En sistemas profesionales, es preferible acumular todos los errores de entrada y devolverlos juntos.

Para ello, se agregan los siguientes métodos:

```
public void agregarError(String mensaje) {
    this.valido = false;
    this.errores.add(mensaje);
}

public void unir(ResultadoValidacion otro) {
    if (!otro.esValido()) {
        this.valido = false;
        this.errores.addAll(otro.getErrores());
    }
}
```

Obtención de un mensaje representativo

En algunos contextos (logs, respuestas simples, pruebas), puede ser útil obtener:

- el primer error, o
- un resumen textual de todos los errores.

```
public String primerError() {
    if (errores.isEmpty()) return "";
    return errores.get(0);
```

```
}

public String resumen() {
    return String.join("; ", errores);
}
```


Versión completa consolidada

Para referencia del lector, el objeto completo queda así:

```
public class ResultadoValidacion {

    private boolean valido;
    private List<String> errores;

    private ResultadoValidacion(boolean valido) {
        this.valido = valido;
        this.errores = new ArrayList<>();
    }

    public static ResultadoValidacion ok() {
        return new ResultadoValidacion(true);
    }

    public static ResultadoValidacion error(String mensaje) {
        ResultadoValidacion r = new ResultadoValidacion(false);
```

```
        r.errores.add(mensaje);
        return r;
    }

    public boolean esValido() {
        return valido;
    }

    public List<String> getErrores() {
        return errores;
    }

    public void agregarError(String mensaje) {
        this.valido = false;
        this.errores.add(mensaje);
    }

    public void unir(ResultadoValidacion otro) {
        if (!otro.esValido()) {
            this.valido = false;
            this.errores.addAll(otro.getErrores());
        }
    }
```

```
    public String primerError() {
        if (errores.isEmpty()) return "";
        return errores.get(0);
    }

    public String resumen() {
        return String.join("; ", errores);
    }
}
```

Clase central propuesta: ValidadorEntrada

A continuación, se propone una clase centralizada (cohesiva y con SRP) que reúne validaciones típicas de entrada. Observa que solo valida y retorna ResultadoValidacion; no hace nada más.

```
class ValidadorEntrada {

    public ResultadoValidacion telefono10Digitos(String telefono) {
        if (telefono == null || telefono.trim().isEmpty()) {
            return ResultadoValidacion.error("El teléfono es obligatorio". );
        }
        if (!telefono.matches("\\d{10}")) {
            return ResultadoValidacion.error("Teléfono inválido. Debe tener 10 dígitos numéricos". );
        }
```

```
        return ResultadoValidacion.ok();
    }

    public ResultadoValidacion enteroEnRango(String texto,
String nombreCampo, int min, int max) {
        if (texto == null || texto.trim().isEmpty()) {
          return ResultadoValidacion.error("El campo '" + nom-
breCampo + "' es obligatorio". );
        }
        String limpio = texto.trim();
        if (!limpio.matches("\\d+")) {
          return ResultadoValidacion.error("El campo '" + nom-
breCampo + "' debe ser un número entero". );
        }
        int valor = Integer.parseInt(limpio);
        if (valor < min || valor > max) {
          return ResultadoValidacion.error("El campo '" + nom-
breCampo + "' debe estar entre " + min + " y " + max + "". );
        }
        return ResultadoValidacion.ok();
    }

    public ResultadoValidacion idSoloDigitosConLongitud(S-
tring id, String nombreCampo, int longitud) {
        if (id == null || id.trim().isEmpty()) {
          return ResultadoValidacion.error("El campo '" + nom-
```

```
breCampo + "' es obligatorio". );
        }
        String limpio = id.trim();
        if (!limpio.matches("\\d+")) {
          return ResultadoValidacion.error("El campo '" + nombreCampo + "' debe contener solo dígitos". );
        }
        if (limpio.length() != longitud) {
              return ResultadoValidacion.error("El campo '" + nombreCampo + "' debe tener exactamente " + longitud + " dígitos". );
        }
        return ResultadoValidacion.ok();
    }

    public ResultadoValidacion requerido(Object valor, String nombreCampo) {
        if (valor == null) {
          return ResultadoValidacion.error("Debe seleccionar un valor para '" + nombreCampo + "'". );
        }
        return ResultadoValidacion.ok();
    }

    public ResultadoValidacion ruc13DigitosSiFactura(boolean requiereFactura, String ruc) {
```

```
        if (!requiereFactura) return ResultadoValidacion.ok();


        if (ruc == null || ruc.trim().isEmpty()) {
          return ResultadoValidacion.error("Si requiere factura,
debe ingresar el RUC". );
        }
        if (!ruc.trim().matches("\\d{13}")) {
              return ResultadoValidacion.error("RUC inválido.
Debe tener 13 dígitos numéricos". );
        }
        return ResultadoValidacion.ok();
    }
}
```

¿Esta clase cumple SRP o se está convirtiendo en "clase Dios"?

En este punto, sí cumple SRP, porque todas sus operaciones pertenecen al mismo concepto: validación de entrada. No está mezclando persistencia, negocio o UI. Sin embargo, existe un riesgo real: si el proyecto crece y se le empieza a agregar lógica de negocio ("no puede retirar más del saldo", "no puede matricularse sin prerrequisito"), entonces dejaría de ser un validador de entrada y se convertiría en una clase sobrecargada. La regla es clara: aquí solo viven reglas de forma, rango y completitud (entrada). Las reglas del dominio se tratan en el epígrafe de negocio.

¿Cómo se integra sin acoplar el servicio?

El servicio (backend) solo invoca el validador y evalúa el resultado. Así se mantiene el acoplamiento bajo: si cambia la validación, el servicio no cambia.

Ejemplo de uso general (patrón de flujo):

```
class ServicioEjemplo {

    private final ValidadorEntrada validador = new Validado-
rEntrada();

    public ResultadoValidacion ejecutarOperacion(String te-
lefono) {
        ResultadoValidacion r = validador.telefono10Digitos(-
telefono);
        if (!r.esValido()) return r;

        // continuar operación normal...
        return ResultadoValidacion.ok();
    }
}
```

Validación de forma vs validación de consistencia dentro de los errores de entrada

Dentro de los errores de entrada no todos los problemas tienen la misma naturaleza. Aunque todos se originan en datos proporcionados por el usuario o por una interfaz externa, es didácticamente útil y profesionalmente necesario distinguir entre dos subtipos de validación: la validación de forma y la validación de consistencia. Esta diferenciación no introduce un nuevo tipo de error, sino que permite comprender mejor qué se está verificando exactamente y por qué sigue siendo responsabilidad de la capa de entrada, incluso cuando intervienen varios campos a la vez.

Validación de forma

La validación de forma verifica que cada dato individual cumpla con requisitos básicos de estructura y representación. Se trata de comprobaciones que pueden realizarse sin conocer el significado profundo del dato en el dominio, y que responden a preguntas como:

- ¿El valor está presente?
- ¿Tiene el tipo correcto?
- ¿Respeta un formato esperado?
- ¿Está dentro de un rango permitido?

Ejemplos típicos de validación de forma son:

- un campo obligatorio no puede ser nulo ni vacío,
- un correo debe cumplir un formato básico,
- una edad debe ser un número positivo dentro de un rango razonable,
- un identificador debe tener una longitud exacta y solo dígitos.

Este tipo de validación es local, directa y predecible. Por esa razón, suele implementarse mediante reglas simples y reutilizables dentro de una clase centralizada de validación, como ValidadorEntrada.

Validación de consistencia

La validación de consistencia, en cambio, verifica que la combinación de varios datos de entrada sea coherente entre sí. A diferencia de la validación de forma, aquí no basta con analizar un campo de manera aislada: es necesario considerar la relación entre dos o más valores.

Ejemplos comunes de validación de consistencia incluyen:

- si se solicita factura, entonces debe proporcionarse un RUC válido,
- la fecha de inicio debe ser anterior a la fecha de fin,
- si el tipo de cliente es “empresa”, entonces el campo razón social es obligatorio,
- si se selecciona un método de pago diferido, deben indicarse cuotas.

Aunque estas validaciones dependen de más de un dato, siguen siendo errores de entrada, porque el problema no es una violación del dominio del negocio, sino una inconsistencia en la información proporcionada. El sistema no está rechazando una operación válida, sino solicitando que los datos de entrada sean completados o corregidos adecuadamente.

¿Por qué esta distinción es importante para el diseño?

No distinguir entre validación de forma y de consistencia conduce a dos errores frecuentes en estudiantes:

- tratar validaciones de consistencia como reglas de negocio, lanzando excepciones innecesarias;
- dispersar estas validaciones en servicios de negocio, mezclando responsabilidades.

Al reconocer explícitamente esta subclasificación, se aprende que:

- ambas validaciones pertenecen a la capa de entrada,
- ambas pueden comunicarse mediante ResultadoValidacion,

- ninguna de ellas debe “romper” el flujo con excepciones.

Este enfoque refuerza la responsabilidad única de las clases de validación, mantiene el dominio limpio y prepara el terreno para comprender cuándo una situación deja de ser una inconsistencia de entrada y se convierte en una verdadera violación de una regla de negocio.

A partir de este punto, los ejemplos de errores de entrada se presentarán indicando explícitamente si corresponden a una validación de forma o a una validación de consistencia. Esta distinción no altera el tipo de error, todos siguen siendo errores de entrada, pero permite al estudiante identificar con claridad qué tipo de comprobación se está realizando y por qué sigue siendo responsabilidad de la capa de validación y no de la lógica de negocio.

Ejemplo 4 (Validación de forma): Número con caracteres inválidos (teléfono)

Situación: El backend recibe un teléfono con letras o símbolos: “09AB-34@56”.

Mala solución (aceptar y guardar tal cual):

```
class UsuarioServiceMal {

    public void registrarUsuario(String nombre, String telefono) {

        // Se guarda sin validar

        System.out.println(“Registrado: “ + nombre + “ Tel: “ + telefono);

    }

}
```

Buena solución (validar en backend sin excepción):

```
class UsuarioService {


    public ResultadoValidacion registrarUsuario(String nombre, String telefono) {

        ResultadoValidacion rTel = validarTelefono(telefono);

        if (!rTel.esValido()) return rTel;


        // Si pasa validación, se continúa con el flujo normal

         System.out.println("Registrado: " + nombre + " Tel: " + telefono);

        return ResultadoValidacion.ok();

    }


    private ResultadoValidacion validarTelefono(String telefono) {

        if (telefono == null || telefono.trim().isEmpty()) {

                return ResultadoValidacion.error("El teléfono es obligatorio". );

        }

        // Ejemplo simple: solo dígitos y longitud 10

        if (!telefono.matches("\\d{10}")) {

            return ResultadoValidacion.error("Teléfono inválido. Debe tener 10 dígitos numéricos". );

        }

        return ResultadoValidacion.ok();
```

```
    }
}
```

Aquí el error es corregible por el usuario (entrada inválida), por eso se devuelve un resultado y no se "revienta" el flujo del programa.

Ejemplo 5 (Validación de forma): Número mal parseado (edad llega como texto)

Situación: El backend recibe "veinte" en un campo numérico.

Mala solución (parsear sin control y dejar que falle)

```
class PersonaServiceMal {
    public void registrar(String edadTexto) {
        int edad = Integer.parseInt(edadTexto); // puede fallar
        System.out.println("Edad registrada: " + edad);
    }
}
```

Buena solución (validar y convertir con resultado)

```
class PersonaService {

    public ResultadoValidacion registrar(String edadTexto) {
        ResultadoValidacion r = validarEdadTexto(edadTexto);
        if (!r.esValido()) return r;

        int edad = Integer.parseInt(edadTexto.trim());
        System.out.println("Edad registrada: " + edad);
```

```
        return ResultadoValidacion.ok();
    }

    private ResultadoValidacion validarEdadTexto(String edadTexto) {
        if (edadTexto == null || edadTexto.trim().isEmpty()) {
            return ResultadoValidacion.error("La edad es obligatoria". );
        }
        if (!edadTexto.trim().matches("\\d+")) {
            return ResultadoValidacion.error("Edad inválida. Debe ingresar un número entero". );
        }

        int edad = Integer.parseInt(edadTexto.trim());
        if (edad < 0 || edad > 120) {
            return ResultadoValidacion.error("Edad fuera de rango. Debe estar entre 0 y 120". );
        }
        return ResultadoValidacion.ok();
    }
}
```

La conversión (parseInt) se hace después de validar. El sistema no cae en excepciones técnicas por datos mal escritos.

Ejemplo 6 (Validación de forma): Selección obligatoria faltante (rol no elegido)

Situación: Se recibe rol = null porque no se seleccionó nada.

Mala solución (asignar rol por defecto sin avisar)

```
enum Rol { ADMIN, DOCENTE, ESTUDIANTE }


class CuentaServiceMal {

   public void crearCuenta(String usuario, Rol rol) {

      if (rol == null) rol = Rol.ESTUDIANTE; // parche silencioso

      System.out.println("Cuenta creada: " + usuario + " Rol: " + rol);

   }

}
```

Buena solución (rechazar entrada con resultado)

```
enum Rol { ADMIN, DOCENTE, ESTUDIANTE }


class CuentaService {


    public ResultadoValidacion crearCuenta(String usuario, Rol rol) {

      ResultadoValidacion r = validarRol(rol);

      if (!r.esValido()) return r;


      System.out.println("Cuenta creada: " + usuario + " Rol: " + rol);
```

```
        return ResultadoValidacion.ok();
    }

    private ResultadoValidacion validarRol(Rol rol) {
        if (rol == null) {
            return ResultadoValidacion.error("Debe seleccionar
un rol (ADMIN, DOCENTE o ESTUDIANTE)". );
        }
        return ResultadoValidacion.ok();
    }
}
```

Un valor obligatorio faltante es un error de entrada. Asignar valores "por defecto" sin consentimiento crea ambigüedad y errores posteriores.

Ejemplo 7 (Validación de forma): Longitud exacta (ID o cédula con tamaño incorrecto)

Situación: Se recibe un identificador con longitud incorrecta: "12345".

Mala solución (truncar o rellenar automáticamente)

```
class IdentificacionServiceMal {
    public String normalizar(String id) {
        if (id.length() > 10) return id.substring(0, 10);
        while (id.length() < 10) id = "0" + id;
        return id;
    }
}
```

Buena solución (validar y exigir corrección)

```
class IdentificacionService {

    public ResultadoValidacion registrarId(String id) {
        ResultadoValidacion r = validarId(id);
        if (!r.esValido()) return r;

        System.out.println("ID registrado: " + id.trim());
        return ResultadoValidacion.ok();
    }

    private ResultadoValidacion validarId(String id) {
        if (id == null || id.trim().isEmpty()) {
            return ResultadoValidacion.error("El ID es obligatorio". );
        }
        String limpio = id.trim();
        if (!limpio.matches("\\d+")) {
            return ResultadoValidacion.error("El ID debe contener solo dígitos". );
        }
        if (limpio.length() != 10) {
            return ResultadoValidacion.error("El ID debe tener exactamente 10 dígitos". );
        }
```

```
        return ResultadoValidacion.ok();
    }
}
```

Un sistema profesional no debe "inventar" identificaciones válidas. Debe exigir datos correctos desde la entrada.

Ejemplo 8 (Validación de consistencia): Datos dependientes (si pide factura, debe venir RUC)

Situación: Si requiereFactura = true, el backend debe recibir ruc. Llega vacío o inválido.

Mala solución (dejar pasar y "resolver después")

```
class FacturacionServiceMal {
    public void procesar(boolean requiereFactura, String ruc)
{
        System.out.println("Procesado. Factura: " + requiere-
Factura + " RUC: " + ruc);
    }
}
```

Buena solución (validación condicional con resultado)

```
class FacturacionService {

    public ResultadoValidacion procesar(boolean requiere-
Factura, String ruc) {
        ResultadoValidacion r = validarFactura(requiereFac-
tura, ruc);
        if (!r.esValido()) return r;
```

```
        System.out.println("Procesado. Factura: " + requiere-
Factura + " RUC: " + (ruc == null ? "" : ruc.trim()));

        return ResultadoValidacion.ok();

    }


    private ResultadoValidacion validarFactura(boolean re-
quiereFactura, String ruc) {

        if (!requiereFactura) return ResultadoValidacion.ok();


        if (ruc == null || ruc.trim().isEmpty()) {

          return ResultadoValidacion.error("Si requiere factura,
debe ingresar el RUC". );

        }

        if (!ruc.trim().matches("\\d{13}")) {

              return ResultadoValidacion.error("RUC inválido.
Debe tener 13 dígitos numéricos". );

        }

        return ResultadoValidacion.ok();

    }

}
```

Aquí no hay "falla del sistema": hay una entrada incompleta. El backend debe detectar la inconsistencia y solicitar corrección, sin caer en excepciones.

Mini demostración de ejecución

```
public class DemoErroresEntrada {

    public static void main(String[] args) {


        UsuarioService usuarioService = new UsuarioService();

        ResultadoValidacion r1 = usuarioService.registrarUsuario("Ana", "09AB-34@56");

        System.out.println("Resultado 1: " + r1.getMensaje());


        PersonaService personaService = new PersonaService();

        ResultadoValidacion r2 = personaService.registrar("veinte");

        System.out.println("Resultado 2: " + r2.getMensaje());


        CuentaService cuentaService = new CuentaService();

        ResultadoValidacion r3 = cuentaService.crearCuenta("miguel", null);

        System.out.println("Resultado 3: " + r3.getMensaje());

    }

}
```

Con estos ejemplos se observa que los errores de entrada pueden manifestarse tanto a nivel de forma individual de los datos como en la coherencia entre varios campos. Reconocer esta diferencia permite diseñar validaciones más claras, reutilizables y correctamente ubicadas, evitando trasladar al dominio problemas que deben resolverse antes de que la operación de negocio comience.

Error de regla de negocio

Los errores de regla de negocio aparecen cuando el sistema recibe datos técnicamente válidos, pero la operación solicitada contradice una norma fundamental del dominio que el software debe proteger. En estos casos, el problema no está en el formato del dato ni en su tipo, sino en su significado dentro del contexto del sistema.

Este tipo de error es especialmente importante porque revela una diferencia clave entre programar y modelar un dominio. El sistema no solo procesa datos, hace cumplir reglas. Cuando una regla de negocio se viola, permitir que el flujo continúe puede dejar el sistema en un estado incoherente o incorrecto.

Ejemplos clásicos de errores de negocio incluyen situaciones como:

- intentar retirar más saldo del disponible en una cuenta bancaria,
- matricularse en una asignatura sin cumplir los prerrequisitos académicos,
- cancelar una orden que ya ha sido enviada al cliente.

A diferencia de los errores de entrada, estos errores no siempre son esperables y, sobre todo, no deben resolverse simplemente informando al usuario y continuando. La operación solicitada no puede completarse, y el sistema debe detener el flujo de forma explícita. Por esta razón, los errores de negocio sí deben representarse mediante excepciones, siempre que estas estén bien diseñadas, tengan un significado claro y expresen directamente la regla violada.

Mala solución: tratar la regla de negocio como un error trivial

```
public class CuentaBancariaMal {

    private double saldo;

    public CuentaBancariaMal(double saldo) {
        this.saldo = saldo;
    }

    public void retirar(double monto) {
        if (monto > saldo) {
            System.out.println("Error");
            return;
        }
        saldo -= monto;
    }
}
```

Problemas de esta solución:

- El mensaje "Error" es ambiguo y no explica la causa real.
- La regla de negocio ("no se puede retirar más del saldo") no queda explícita en el diseño.
- El flujo se interrumpe de forma silenciosa.

- En sistemas grandes, este error puede pasar desapercibido o ser ignorado.
- No hay forma clara de que capas superiores reaccionen adecuadamente.

Desde el punto de vista pedagógico, esta solución enseña una mala práctica: ocultar la gravedad de una violación del dominio.

Buena solución: expresar la regla de negocio mediante una excepción propia

Definición de la excepción de negocio:

```
public class ReglaNegocioException extends RuntimeException {


    public ReglaNegocioException(String mensaje) {
        super(mensaje);
    }
}
```

Aplicación correcta en la lógica de negocio:

```
public class CuentaBancaria {


    private double saldo;


    public CuentaBancaria(double saldo) {
        this.saldo = saldo;
    }
```

```
    public void retirar(double monto) {
        if (monto > saldo) {
            throw new ReglaNegocioException(
                "Saldo insuficiente para realizar el retiro. Saldo
actual: " + saldo
            );
        }
        saldo -= monto;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

Ventajas de esta solución:

- La regla del dominio queda claramente expresada.
- El flujo se interrumpe correctamente.
- La excepción puede ser capturada y gestionada por capas superiores.
- El sistema se protege contra estados inválidos.
- El código comunica intención, no solo control de flujo.

Una regla de negocio violada no se negocia ni se silencia: se expresa con claridad y se impone deteniendo el proceso.

Error técnico o del sistema

Los errores técnicos son aquellos que no dependen del usuario ni del dominio del negocio, sino de fallos de infraestructura, recursos externos o componentes internos del sistema. A diferencia de los errores de entrada o de negocio, estos errores suelen estar fuera del control inmediato del programador y del usuario final.

Ejemplos frecuentes de errores técnicos incluyen:

- intentar acceder a un archivo que no existe,
- fallos de conexión a una base de datos,
- servicios externos que no responden,
- errores de red o de configuración del entorno.

Desde un punto de vista didáctico y profesional, es fundamental comprender que los errores técnicos nunca deben ocultarse. Silenciarlos o reemplazarlos por mensajes genéricos produce sistemas imposibles de depurar, dificulta el mantenimiento y puede provocar fallos graves en producción.

Mala solución: ocultar el error técnico

```
public class LectorArchivoMal {


    public void leerArchivo() {
        try {
            // Supongamos que aquí se intenta leer un archivo
            throw new IOException("Archivo no encontrado");
        } catch (Exception e) {
            System.out.println("Algo salió mal");
```

```
        }
    }
}
```

Problemas de esta solución:

- Se pierde completamente la causa real del error.
- No queda información para depuración o registro.
- El sistema aparenta continuar "normalmente" cuando no debería.
- En producción, este tipo de código genera los llamados bugs fantasma.

Esta práctica enseña al estudiante a esconder problemas, no a resolverlos.

Buena solución: propagar o traducir el error técnico preservando la causa

```
public class LectorArchivo {


    public void leerArchivo() {
        try {
            // Simulación de lectura de archivo
                throw new IOException("Archivo de configuración
no encontrado");
        } catch (IOException e) {
            throw new RuntimeException(
                "No se pudo leer el archivo de configuración del
sistema", e
            );
```

```
        }
    }
}
```

Ventajas de esta solución:

- El error se propaga correctamente.
- Se conserva la causa original (IOException).
- Permite que capas superiores tomen decisiones adecuadas: reintentar, mostrar mensaje técnico, registrar el error, detener el sistema.
- Facilita el diagnóstico y mantenimiento.

Un error técnico no se corrige con validaciones ni reglas; se propaga o se traduce, pero nunca se esconde.

Anti-patrones clásicos en validación (y por qué destruyen el diseño)

Después de distinguir error de entrada, error de negocio y error técnico, el siguiente paso es reconocer un fenómeno común: muchos sistemas fallan no por desconocer la teoría, sino por aplicarla mal en decisiones pequeñas y repetidas. En validación, esos errores se convierten en anti-patrones: soluciones "rápidas" que funcionan una vez, pero que, cuando el sistema crece, generan duplicación, inconsistencias, condicionales dispersos y fallos difíciles de rastrear.

En un libro de programación orientada a objetos, estudiar anti-patrones no es opcional: es la manera más directa de que el estudiante aprenda a pensar como diseñador y no como "escritor de if". Cada anti-patrón que verás a continuación se presenta con situación realista, código ingenuo (mala práctica), problema estructural, reconstrucción (buena práctica) usando lo

ya aprendido (SRP, cohesión, bajo acoplamiento y el enfoque de ResultadoValidacion para entrada).

Anti-patrón 1: Validar en la UI y validar distinto en el servicio (inconsistencia de reglas)

Situación: Un formulario web valida el correo con una regla "simple", pero el backend valida con otra regla diferente. Ambos creen tener la verdad. En producción, aparecen casos donde "a veces pasa, a veces no", dependiendo de por dónde entró el dato.

Mala práctica (reglas duplicadas e inconsistentes)

```
// UI (por ejemplo, formulario)
if (!correo.contains("@")) {
    mostrar("Correo inválido");
    return;
}

// Servicio (backend)
if (correo.length() < 10 || !correo.endsWith(".com")) {
    return ResultadoValidacion.error("Correo inválido");
}
```

Problema estructural

- La regla está duplicada en dos lugares.
- Peor: está diferente, así que el sistema se vuelve impredecible.
- Cuando cambie la regla, alguien actualizará un lado y olvidará el otro.

- El estudiante aprende lo contrario a “una fuente de verdad”.

Buena práctica (validación centralizada consumida por todas las capas)

```
class ValidadorEntrada {


   public ResultadoValidacion correoBasico(String correo) {
      if (correo == null || correo.trim().isEmpty()) {
         return ResultadoValidacion.error(“El correo es obligatorio”. );
      }
      String c = correo.trim();
      if (!c.contains(“@”) || !c.contains(“”. )) {
         return ResultadoValidacion.error(“Correo inválido. Debe contener ‘@’ y un dominio”. );
      }
      return ResultadoValidacion.ok();
   }
}


class UsuarioService {
   private final ValidadorEntrada validador = new ValidadorEntrada();


   public ResultadoValidacion registrar(String correo) {
     ResultadoValidacion r = validador.correoBasico(correo);
```

```
        if (!r.esValido()) return r;


        // continuar operación...
        return ResultadoValidacion.ok();
    }
}
```


La UI puede mostrar mensajes amigables, pero la validación real debe poder ejecutarse siempre en el backend (porque el backend protege el sistema). La UI solo "ayuda"; el backend garantiza.

Anti-patrón 2: Validar después de persistir (validación tardía)

Situación: El estudiante guarda primero y valida después "porque así ya tiene el ID" o "porque es más fácil". El resultado es almacenar datos inválidos y luego intentar corregirlos.

Mala práctica

```
public void guardarUsuario(String nombre) {
    repositorio.insertar(nombre); // ya quedó guardado...
    if (nombre == null || nombre.trim().isEmpty()) {
        System.out.println("Nombre inválido");
    }
}
```

Problema estructural

- El sistema ya quedó en estado inválido.
- Si luego falla algo, la causa aparece "tarde".

- Se rompe el principio: primero se verifica, luego se actúa.

Buena práctica (validar antes; persistir solo si es válido)

```
public ResultadoValidacion guardarUsuario(String nombre)
{
   if (nombre == null || nombre.trim().isEmpty()) {
      return ResultadoValidacion.error("El nombre es obligatorio". );
   }
   repositorio.insertar(nombre.trim());
   return ResultadoValidacion.ok();
}
```

Criterio: la validación de entrada funciona como "puerta". Si la puerta está abierta (válido), recién se permite modificar el estado del sistema.

Anti-patrón 3: "Arreglar" datos silenciosamente (normalización abusiva)

Situación: El estudiante "corrige" una edad negativa convirtiéndola en 0, o recorta un ID a 10 dígitos, o rellena con ceros. El sistema "funciona", pero pierde verdad.

Mala práctica (parches que ocultan errores)

```
if (edad < 0) edad = 0; // "lo arreglo"
```

Problema estructural

- Se pierde trazabilidad: nadie sabe que hubo un error.
- Se fabrican datos falsos, que luego contaminan reportes.

- En casos legales/financieros, esto es gravísimo.

Buena práctica (rechazar con claridad, sin inventar datos)

```
if (edad < 0 || edad > 120) {
    return ResultadoValidacion.error("Edad inválida. Debe estar entre 0 y 120". );
}
```

La normalización solo es aceptable cuando no cambia el significado del dato (por ejemplo, trim() en textos). Cambiar valores (-5 a 0) cambia el significado: eso no es normalizar, es falsear.

Anti-patrón 4: Convertir reglas de negocio en ResultadoValidacion (mezclar entrada con dominio)

Situación: Se usa ResultadoValidacion para decir "saldo insuficiente" o "pedido no cancelable", tratando una violación del dominio como si fuera "un formulario mal llenado".

Mala práctica (dominio como si fuera entrada)

```
public ResultadoValidacion retirar(double monto) {
    if (monto > saldo) {
        return ResultadoValidacion.error("Saldo insuficiente");
    }
    saldo -= monto;
    return ResultadoValidacion.ok();
}
```

Problema estructural

- La operación no debería continuar: es una regla del negocio.

- Se obliga a que cada capa recuerde revisar el resultado.
- Un programador puede ignorarlo accidentalmente y seguir el flujo igual.
- La regla pierde "peso" conceptual: parece un error corregible, cuando no lo es.

Buena práctica (negocio = excepción semántica que impone la regla)

```
public void retirar(double monto) {
    if (monto > saldo) {
        throw new SaldoInsuficienteException(saldo, monto);
    }
    saldo -= monto;
}
```

Criterio:

Entrada inválida → se corrige → ResultadoValidacion.

Regla de negocio violada → se impone → excepción de negocio.

Anti-patrón 5: Usar excepciones para errores de entrada "en masa" (try-catch como control de flujo)

Situación: Se pone todo en try-catch y lanza excepciones por campo vacío, por formato mal, por rango. Luego envuelve todo con un catch genérico.

Mala práctica (excepciones como validación normal)

```
public void registrar(String edadTexto) {
    try {
        if (edadTexto == null) throw new RuntimeException("E-
dad requerida");
```

```
        int edad = Integer.parseInt(edadTexto);
        if (edad < 0) throw new RuntimeException("Edad inválida");
        // ...
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

Problema estructural

- Convierte casos esperables en "fallos".
- Hace que el flujo sea difícil de leer.
- Obliga a manejar demasiadas rutas con try-catch.
- Enseña una mala idea: "programar a golpes de excepción".

Buena práctica (validación explícita + acumulable si aplica)

```
public ResultadoValidacion registrar(String edadTexto) {
    if (edadTexto == null || edadTexto.trim().isEmpty()) {
        return ResultadoValidacion.error("La edad es obligatoria". );
    }
    if (!edadTexto.trim().matches("\\d+")) {
        return ResultadoValidacion.error("Edad inválida. Debe ser un entero". );
    }
    int edad = Integer.parseInt(edadTexto.trim());
```

```
        if (edad < 0 || edad > 120) {
            return ResultadoValidacion.error("Edad fuera de rango
    (0 a 120)". );
        }
        return ResultadoValidacion.ok();
    }
```


Idea clave: si el usuario puede corregirlo, no es "excepcional"; es parte normal del flujo.

En este epígrafe se ha establecido una distinción fundamental entre los distintos tipos de errores que pueden aparecer en un sistema de software: errores de entrada, errores de regla de negocio y errores técnicos. Esta clasificación no es meramente teórica, sino que determina cómo debe reaccionar el sistema ante cada situación y qué mecanismos de diseño resultan más adecuados en cada caso.

Hasta aquí, el lector aprendió a gestionar entradas inválidas de manera controlada, evitando que el sistema se llene de condicionales dispersos o de excepciones usadas como validación. Sin embargo, cuando los datos son correctos y aun así la operación viola una regla del dominio, el tratamiento debe cambiar: ya no se trata de "corregir", sino de proteger la coherencia del negocio. En el epígrafe 5.3 se desarrollará cómo diseñar y centralizar excepciones de negocio para expresar reglas con claridad, detener procesos cuando corresponde y permitir que capas superiores gestionen el error de forma profesional.

5.2. Manejo de errores de negocio con excepciones propias

Un error de negocio se produce cuando una acción solicitada contradice una regla esencial del dominio que el sistema representa. A diferencia de los errores

de entrada, estos errores no se corrigen ajustando un dato, sino rechazando la operación completa. El sistema no debe "adaptarse" ni "buscar alternativas", sino hacer cumplir la norma.

Desde el punto de vista del diseño orientado a objetos, los errores de negocio tienen un rol clave: definen los límites del comportamiento permitido del sistema. Por ello, su manejo debe ser explícito, semántico y consistente. Tratar una violación de negocio como un simple if con un mensaje genérico degrada el modelo y oculta la intención del código.

Aquí aparece un principio central: Las reglas de negocio no se validan, se imponen.

Imponer una regla implica detener el flujo cuando esta se viola, y el mecanismo natural para ello en Java es el uso de excepciones propias de negocio.

¿Por qué usar excepciones propias y no excepciones genéricas?

Una mala práctica frecuente consiste en reutilizar excepciones genéricas (RuntimeException, IllegalArgumentException) para representar violaciones de negocio. Aunque técnicamente funciona, esta estrategia tiene serias desventajas pedagógicas y de diseño:

- No expresa claramente la regla violada.
- Obliga a interpretar mensajes de texto para entender el problema.
- Dificulta el manejo diferenciado de errores en capas superiores.
- Mezcla errores técnicos con errores del dominio.

La solución correcta consiste en crear excepciones que formen parte del lenguaje del dominio, de modo

que el código se lea casi como una especificación del negocio.

Centralización de excepciones de negocio

Así como se centralizó la validación de errores de entrada, las excepciones de negocio también deben centralizarse mediante una jerarquía clara, evitando su definición dispersa en múltiples clases.

Clase base para excepciones de negocio

```
public abstract class ExcepcionNegocio extends Runtime-
Exception {


   public ExcepcionNegocio(String mensaje) {
      super(mensaje);
   }
}
```

Justificación de diseño:

- Representa explícitamente el concepto “error de negocio”.
- Permite capturar todas las excepciones del dominio con un solo tipo base.
- Facilita el manejo uniforme en capas superiores (controladores, servicios).
- Evita confundir reglas de negocio con fallos técnicos.

Esta clase no contiene lógica, solo semántica. Su propósito es estructural, no operativo.

Excepciones de negocio específicas: expresar las reglas

Ejemplo 1: Saldo insuficiente

```
public class SaldoInsuficienteException extends ExcepcionNegocio {


    public SaldoInsuficienteException(double saldo, double monto) {
        super("Saldo insuficiente. Saldo actual: " + saldo +
            ", monto solicitado: " + monto);
    }
}
```

Ejemplo 2: Prerrequisito no cumplido

```
public class PrerrequisitoNoCumplidoException extends ExcepcionNegocio {


    public PrerrequisitoNoCumplidoException(String asignatura) {
        super("No cumple los prerrequisitos para matricularse en: " + asignatura);
    }
}
```

Estas clases no son "errores técnicos": son reglas explícitas del dominio, expresadas como tipos propios.

Precondiciones técnicas vs reglas del dominio

Antes de analizar el uso concreto de excepciones en los servicios de negocio, es necesario establecer una distinción fundamental que suele generar confusión en los estudiantes: no todas las excepciones lanzadas dentro de un método representan reglas de negocio. Algunas excepciones existen únicamente para proteger la correcta invocación técnica del método, mientras que otras expresan violaciones explícitas del dominio.

En programación orientada a objetos, toda operación define implícitamente un conjunto de precondiciones técnicas: condiciones mínimas que deben cumplirse para que el método tenga sentido desde el punto de vista computacional. Por ejemplo, un monto a retirar debe ser un número positivo; un identificador no puede ser nulo; una lista no puede ser vacía cuando se espera procesar elementos. Estas condiciones no pertenecen al dominio, sino al contrato técnico del método. Cuando se violan, lo correcto es utilizar excepciones técnicas estándar, como IllegalArgumentException, que indican un uso incorrecto de la operación.

En contraste, las reglas de negocio representan restricciones propias del dominio que el sistema modela. Estas reglas pueden ser violadas incluso cuando los datos son técnicamente válidos. Retirar un monto positivo mayor al saldo disponible, matricularse sin prerrequisitos o confirmar un pedido ya enviado son ejemplos claros de violaciones del dominio. En estos casos, el error no está en cómo se invoca el método, sino en lo que se intenta hacer dentro del modelo del negocio, y por tanto deben representarse mediante excepciones propias del dominio.

Este criterio permite mantener un diseño claro y coherente:

- Las precondiciones técnicas protegen al método de usos inválidos.
- Las excepciones de negocio protegen la integridad del dominio.

Mezclar ambos conceptos sin distinción conduce a código confuso y a modelos pobres que no reflejan correctamente las reglas del sistema.

Uso correcto en servicios de negocio

Ejemplo: Servicio bancario

```
public class CuentaBancaria {


   private double saldo;


   public CuentaBancaria(double saldoInicial) {
      this.saldo = saldoInicial;
   }


   public void retirar(double monto) {
      if (monto <= 0) {
        throw new IllegalArgumentException("El monto debe
ser positivo". );
      }
      if (monto > saldo) {
        throw new SaldoInsuficienteException(saldo, monto);
```

```
        }
        saldo -= monto;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

Análisis:

- El monto es técnicamente válido.
- La violación ocurre en la regla del negocio, no en la entrada.
- La excepción detiene el proceso de forma clara.
- El código expresa intención, no solo control de flujo.

Beneficios del enfoque estructurado

Adoptar este enfoque aporta múltiples beneficios:

- El código refleja fielmente las reglas del dominio.
- Se evita la proliferación de condicionales dispersos.
- El sistema se vuelve más robusto y predecible.
- Las capas superiores pueden manejar errores de forma uniforme.
- El diseño escala sin perder claridad.

Desde una perspectiva formativa, este epígrafe enseña a pensar en términos de dominio, no solo de

instrucciones, consolidando una de las competencias más importantes del desarrollo profesional.

Catálogo de excepciones de negocio: construir un lenguaje explícito del dominio

En sistemas orientados a objetos bien diseñados, las excepciones de negocio no son un recurso técnico improvisado, sino una herramienta semántica que permite expresar con claridad las reglas fundamentales del dominio. Una excepción de negocio bien nombrada y correctamente estructurada funciona como una frase del lenguaje del sistema: describe qué regla fue violada, en qué contexto y por qué la operación no puede continuar.

Desde una perspectiva formativa, este punto es crítico. Muchos estudiantes comprenden que "hay que lanzar excepciones", pero no saben cómo nombrarlas, qué información incluir ni cómo organizarlas. El resultado suele ser un conjunto caótico de excepciones genéricas (RuntimeException, Exception) con mensajes ambiguos, que no aportan claridad ni ayudan al mantenimiento del sistema.

Este epígrafe propone un catálogo orientativo de excepciones de negocio, no como una lista cerrada, sino como un modelo de diseño que el estudiante puede adaptar a distintos dominios. El objetivo no es memorizar nombres, sino aprender a construir excepciones que comuniquen intención y protejan el dominio.

Principios para diseñar excepciones de negocio

Antes de presentar ejemplos concretos, es importante establecer algunos criterios generales que deben cumplirse siempre que se diseñe una excepción de negocio:

- La excepción debe expresar una regla del dominio, no un fallo técnico.
- El nombre de la excepción debe ser autoexplicativo, sin necesidad de leer el mensaje.
- El mensaje debe ser claro, contextual y orientado al dominio, no al programador.
- La excepción puede incluir datos relevantes del contexto, si eso ayuda a entender el error.
- Todas las excepciones de negocio deben compartir una clase base común, para facilitar su gestión.

Estos principios evitan que las excepciones se conviertan en simples "mensajes con throw" y las transforman en parte del modelo conceptual del sistema.

Estructura base para excepciones de negocio

Tal como se introdujo en epígrafes anteriores, todas las excepciones de negocio deben heredar de una clase base común, que represente explícitamente el concepto "error de negocio".

```
public abstract class ExcepcionNegocio extends Runtime-
Exception {


   public ExcepcionNegocio(String mensaje) {

      super(mensaje);

   }

}
```

Esta clase no contiene lógica adicional. Su valor no está en lo que hace, sino en lo que representa: cualquier

excepción que herede de ella pertenece al dominio y no a la infraestructura técnica.

Excepciones de negocio frecuentes: ejemplos y justificación

A continuación, se presentan ejemplos típicos de excepciones de negocio, organizados por tipo de dominio. Cada ejemplo incluye la justificación de su existencia y las decisiones de diseño involucradas.

Ejemplo 1: SaldoInsuficienteException

```
public class SaldoInsuficienteException extends ExcepcionNegocio {


    public SaldoInsuficienteException(double saldo, double monto) {
        super("Saldo insuficiente. Saldo actual: " + saldo +
            ", monto solicitado: " + monto);
    }
}
```

Justificación de diseño

- El nombre expresa claramente la regla violada: no se puede retirar más de lo disponible.
- No es un error de entrada: el monto puede ser positivo y válido.
- No es un error técnico: el sistema funciona correctamente.
- Es una violación del dominio financiero, por lo que debe detener el flujo.

Incluir el saldo y el monto en el mensaje aporta contexto, lo que facilita la comprensión del problema y el diagnóstico.

Ejemplo 2: PedidoNoCancelableException

```
public class PedidoNoCancelableException extends ExcepcionNegocio {


    public PedidoNoCancelableException(String estadoActual) {

        super("El pedido no puede cancelarse porque su estado actual es: " + estadoActual);

    }

}
```

Justificación de diseño

- La regla del dominio es clara: ciertos estados no permiten cancelación.
- El nombre evita ambigüedad y elimina la necesidad de comentarios.
- El estado actual se incluye para explicar por qué la operación es inválida.

Esta excepción refuerza la idea de que el estado del objeto es parte del dominio, no un detalle técnico.

Ejemplo 3: PrerrequisitoNoCumplidoException

```
public class PrerrequisitoNoCumplidoException extends ExcepcionNegocio {


    public PrerrequisitoNoCumplidoException(String asignatura) {
```

```
        super("No cumple los prerrequisitos para matricularse
    en la asignatura: " + asignatura);
    }
}
```

Justificación de diseño

- La excepción pertenece claramente al dominio académico.
- El mensaje es comprensible incluso para alguien que no vea el código.
- Evita soluciones débiles como "return false" o mensajes genéricos.

Este tipo de excepción enseña al estudiante a modelar reglas institucionales, no solo a controlar flujo.

Ejemplo 4: CupoAgotadoException

```
public class CupoAgotadoException extends ExcepcionNe-
gocio {

    public CupoAgotadoException(String codigoAsignatura) {
        super("No existen cupos disponibles para la asignatura:
    " + codigoAsignatura);
    }
}
```

Justificación de diseño

- El cupo no es un detalle técnico; es una restricción del dominio.
- El sistema debe impedir la operación, no sugerir correcciones.

- La excepción protege la coherencia del sistema académico.

Sobre la nomenclatura de excepciones de negocio

Un error común es utilizar nombres vagos como:

- OperacionInvalidaException
- NegocioException
- ErrorSistemaException

Estos nombres no comunican intención. En cambio, una buena excepción de negocio:

- comienza con un sustantivo del dominio (Saldo, Pedido, Cupo, Prerrequisito),
- describe claramente la condición inválida,
- termina con Exception por convención.

Este estilo convierte al código en una forma de documentación viva del dominio.

¿Qué no debe contener una excepción de negocio?

- No debe contener lógica de negocio.
- No debe acceder a bases de datos ni servicios externos.
- No debe capturar otras excepciones técnicas.
- No debe usarse para validar entrada de usuario.

Su único propósito es detener el flujo cuando una regla del dominio se viola y comunicar claramente esa violación.

Relación con el diseño orientado a objetos

- Responsabilidad única: cada excepción representa una regla específica.

- Alta cohesión: la excepción existe solo para expresar una condición del dominio.
- Bajo acoplamiento: las capas superiores pueden manejar todas las excepciones de negocio de forma uniforme gracias a la clase base común.
- Claridad semántica: el código se lee como una especificación del sistema.

Una vez que las reglas del dominio se expresan mediante excepciones claras y bien estructuradas, el sistema gana coherencia y legibilidad. Sin embargo, aún queda un aspecto fundamental por abordar: los errores que no pertenecen al dominio ni al usuario, sino a la infraestructura técnica. En el siguiente epígrafe se profundizará en cómo traducir y propagar correctamente estos errores entre capas, evitando que detalles técnicos contaminen la lógica del negocio y manteniendo la trazabilidad necesaria para sistemas profesionales.

5.3. Manejo de errores técnicos y estrategias de propagación

Los errores técnicos o del sistema son aquellos que se producen como consecuencia de fallos en la infraestructura, en los recursos externos o en componentes internos que no forman parte del dominio del negocio. A diferencia de los errores de entrada y de las reglas de negocio, estos errores no representan una decisión incorrecta del usuario ni una violación de normas del sistema, sino una incapacidad técnica para completar una operación.

Ejemplos habituales de errores técnicos incluyen:

- archivos inexistentes o inaccesibles,
- fallos de conexión a bases de datos,
- servicios externos no disponibles,

- errores de red,
- configuraciones incorrectas del entorno de ejecución.

Desde el punto de vista del diseño, estos errores presentan un desafío particular: no pueden ser "arreglados" por el sistema en el momento, y tampoco deben confundirse con reglas del dominio. Su tratamiento incorrecto conduce a sistemas frágiles, opacos y extremadamente difíciles de depurar.

El error técnico como información crítica

Uno de los errores conceptuales más comunes en programadores novatos es ocultar los errores técnicos, ya sea capturándolos sin hacer nada o sustituyéndolos por mensajes genéricos. Este enfoque genera la ilusión de estabilidad, pero en realidad produce sistemas inseguros y poco confiables.

Un error técnico debe cumplir tres principios fundamentales:

- No debe perderse la causa original.
- Debe poder ser diagnosticado.
- Debe ser gestionado en el nivel adecuado.

Por esta razón, los errores técnicos se propagan o se traducen, pero nunca se silencian.

Mala práctica: captura indiscriminada

```
public class ConfiguracionMal {


    public void cargar() {
        try {
            // Simulación de lectura de archivo
```

```
            throw new IOException("Archivo no encontrado");
        } catch (Exception e) {
            System.out.println("Error al cargar configuración");
        }
    }
}
```


Problemas de este enfoque:

- Se pierde la causa real del fallo.
- El sistema continúa en un estado potencialmente inválido.
- No existe información útil para depuración.
- Se generan los llamados bugs fantasma.

Buena práctica: propagación controlada

```
public class Configuracion {


    public void cargar() {
        try {
            // Simulación de lectura de archivo
             throw new IOException("Archivo de configuración
no encontrado");
        } catch (IOException e) {
            throw new RuntimeException(
              "No se pudo cargar la configuración del sistema", e
            );
```

```
        }
    }
}
```

Ventajas:

- Se preserva la excepción original (IOException).
- El error puede ser registrado y diagnosticado.
- Capas superiores pueden decidir:
- abortar la ejecución,
- reintentar,
- mostrar un mensaje técnico.

Un error técnico no se oculta: se propaga con contexto.

Traducción vs propagación

En sistemas bien diseñados, no siempre es deseable propagar literalmente una excepción técnica. En ocasiones, conviene traducirla a una excepción más acorde al nivel de abstracción.

```
public class RepositorioUsuarios {


    public void guardar() {
        try {
            throw new SQLException("Base de datos no disponible");
        } catch (SQLException e) {
            throw new RuntimeException("Error al acceder a datos de usuarios", e);
        }
```

```
        }
    }
```

Este enfoque evita exponer detalles técnicos innecesarios a capas que no los necesitan, sin perder la trazabilidad.

Traducción de errores técnicos por capa: del detalle técnico a la decisión arquitectónica

Hasta este punto del capítulo se ha establecido que los errores técnicos no deben ocultarse y que su causa original debe preservarse. Sin embargo, en sistemas orientados a objetos de tamaño medio o grande, no basta con propagar excepciones técnicas sin más. Es necesario decidir en qué capa se traduce el error, qué nivel de detalle se conserva y quién es responsable de comunicarlo.

Este epígrafe introduce una práctica fundamental de diseño profesional: la traducción de errores técnicos por capa. El objetivo no es "atrapar excepciones", sino mantener las responsabilidades claras y evitar que detalles de infraestructura contaminen capas que no deberían conocerlos.

Principio general: cada capa conoce solo lo que necesita

Un sistema orientado a objetos bien estructurado suele organizarse, al menos conceptualmente, en capas:

- Repositorio (o infraestructura): acceso a datos, archivos, red.
- Servicio (o aplicación): orquesta casos de uso.
- Controlador (o frontera): interactúa con el usuario o interfaz externa.

Cada capa tiene un nivel de abstracción distinto, y por tanto, un nivel distinto de conocimiento técnico. Permitir

que una SQLException llegue hasta el controlador, o peor aún, hasta la interfaz de usuario, rompe este principio y acopla el sistema a detalles que no le corresponden.

Capa de repositorio: capturar y traducir errores de infraestructura

El repositorio es la capa más cercana a la infraestructura. Aquí es normal trabajar con excepciones técnicas específicas como SQLException, IOException, etc. Sin embargo, estas excepciones no deben escapar directamente.

Mala práctica: propagar la excepción técnica cruda

```
public class UsuarioRepositorioMal {


   public void guardar(Usuario u) throws SQLException {

      // acceso directo a base de datos

       throw new SQLException("Base de datos no disponible");

   }

}
```

Problemas

- Obliga a todas las capas superiores a conocer SQLException.
- Acopla el dominio a una tecnología concreta.
- Hace el sistema frágil ante cambios de infraestructura.

Buena práctica: traducir a una excepción técnica propia

```
public class ErrorPersistenciaException extends Runtime-
```

```
Exception {


   public ErrorPersistenciaException(String mensaje, Throw-
able causa) {
       super(mensaje, causa);
   }
}


public class UsuarioRepositorio {


   public void guardar(Usuario u) {
       try {
          // acceso a base de datos
           throw new SQLException("Base de datos no dispo-
nible");
       } catch (SQLException e) {
          throw new ErrorPersistenciaException(
                "No se pudo guardar el usuario en la base de
datos", e
          );
       }
   }
}
```

Justificación

- Se preserva la causa original (SQLException).

- Se elimina la dependencia directa de la tecnología.
- El repositorio comunica qué falló, no cómo falló.

Capa de servicio: decidir si traduce o propaga

El servicio no es infraestructura, pero tampoco es interfaz. Su responsabilidad es orquestar el caso de uso. Aquí aparece una decisión de diseño importante:

¿Debe el servicio traducir de nuevo la excepción o dejarla pasar?

Opción A: Propagar tal como viene (válido en sistemas pequeños)

```
public class UsuarioService {


    private final UsuarioRepositorio repo = new UsuarioRe-
positorio();


   public void registrar(Usuario u) {

       repo.guardar(u); // ErrorPersistenciaException puede
propagarse

   }

}
```

Esta opción es aceptable cuando:

- el sistema es pequeño,
- no hay múltiples fuentes técnicas,
- la excepción ya es suficientemente abstracta.

Opción B: Traducir a una excepción de infraestructura más general

```
public class ErrorInfraestructuraException extends RuntimeException {

    public ErrorInfraestructuraException(String mensaje, Throwable causa) {
        super(mensaje, causa);
    }
}

public class UsuarioService {

    private final UsuarioRepositorio repo = new UsuarioRepositorio();

    public void registrar(Usuario u) {
        try {
            repo.guardar(u);
        } catch (ErrorPersistenciaException e) {
            throw new ErrorInfraestructuraException(
                "Error técnico al registrar el usuario", e
            );
        }
    }
}
```

Criterio

- El servicio no entra en detalles técnicos.
- Comunica: “el caso de uso no pudo completarse por un problema técnico”.
- Mantiene bajo acoplamiento entre capas.

Capa de controlador: comunicar sin ocultar

El controlador es el borde del sistema. Aquí sí tiene sentido capturar excepciones generales, porque su función es decidir cómo se comunica el problema, no resolverlo.

Ejemplo de controlador bien diseñado

```
public class UsuarioController {


   public void crearUsuario(Usuario u) {


      UsuarioService service = new UsuarioService();


      try {
         service.registrar(u);
         System.out.println(“Usuario registrado correctamente”. );


      } catch (ExcepcionNegocio e) {
            System.out.println(“Operación inválida: “ + e.getMessage());
```

```
        } catch (ErrorInfraestructuraException e) {
            System.out.println("Error técnico. Intente más tarde".
);
            e.printStackTrace(); // logging real en sistemas pro-
fesionales
        }
    }
}
```


Análisis

- El usuario recibe un mensaje comprensible.
- El detalle técnico no se expone al usuario final.
- La causa original no se pierde.
- El controlador no "arregla" el error; solo lo comunica.

¿Qué se logra con la traducción por capa?

Este enfoque aporta beneficios claros:

- Separación de responsabilidades: cada capa hace lo que le corresponde.
- Bajo acoplamiento: las capas superiores no dependen de detalles técnicos.
- Escalabilidad: cambiar base de datos o tecnología no rompe el dominio.
- Trazabilidad: la causa original siempre está disponible.
- Claridad conceptual: el código refleja la arquitectura del sistema.

Desde el punto de vista formativo, este epígrafe enseña algo crucial, manejar errores no es "atraparlos", es diseñar su recorrido por el sistema.

Errores frecuentes que este enfoque evita

- Capturar Exception en repositorios.
- Mostrar mensajes técnicos al usuario.
- Repetir try-catch idénticos en todas las capas.
- Mezclar reglas de negocio con fallos técnicos.
- Perder la causa original del error.

Uso de try-with-resources: gestión segura de recursos

En el manejo de errores técnicos, uno de los problemas más frecuentes es la incorrecta liberación de recursos como archivos, conexiones a bases de datos o flujos de red. En diseños ingenuos, el programador suele cerrar estos recursos manualmente dentro de bloques finally, lo cual es propenso a errores y omisiones.

El mecanismo try-with-resources fue introducido precisamente para resolver este problema de forma segura y expresiva. Su principal ventaja es que garantiza el cierre automático del recurso, incluso cuando ocurre una excepción, eliminando una fuente común de fugas de memoria y bloqueos de sistema.

```
try (BufferedReader reader = new BufferedReader(new
FileReader("config.txt"))) {
   return reader.readLine();
} catch (IOException e) {
    throw new RuntimeException("Error al leer configura-
ción", e);
}
```

Desde el punto de vista del diseño, try-with-resources no solo mejora la seguridad técnica, sino que hace explícita la responsabilidad del método: usar un recurso y liberarlo correctamente, sin delegar esa tarea al programador de forma manual.

Checked vs unchecked: cuándo traducir excepciones

Java distingue entre excepciones checked y unchecked. Las primeras obligan a ser declaradas o capturadas, mientras que las segundas no. En sistemas profesionales, esta distinción debe usarse con criterio y no de forma automática.

En capas de infraestructura (acceso a archivos, base de datos, red), las excepciones checked son apropiadas porque describen fallos técnicos concretos. Sin embargo, propagarlas directamente a capas superiores suele contaminar el diseño, obligando a que el dominio conozca detalles técnicos irrelevantes.

```
try {
    // acceso a base de datos
} catch (SQLException e) {
    throw new RuntimeException("Error de acceso a datos", e);
}
```

Esta traducción permite preservar la causa original sin forzar a todas las capas a depender de excepciones técnicas específicas.

Reintentos (retry): ¿cuándo aplicar y cuándo NO hacerlo?

Un error frecuente en sistemas distribuidos es asumir que todo error técnico debe reintentarse. Esta suposición es peligrosa. Reintentar sin criterio puede

agravar el problema, saturar recursos y ocultar fallos graves.

Los reintentos solo son apropiados cuando:

- el fallo es transitorio,
- no produce efectos secundarios,
- existe un límite claro de intentos.

No deben aplicarse cuando:

- la operación no es idempotente,
- el fallo es persistente,
- se desconoce la causa del error.

Timeouts: fallar rápido es una virtud

En sistemas reales, esperar indefinidamente por un recurso externo es una de las causas más graves de bloqueos. Por ello, un diseño robusto debe contemplar el concepto de timeout. Un timeout no es un error del dominio ni del usuario; es una condición técnica que indica que el sistema no pudo obtener una respuesta en un tiempo razonable. Diseñar para fallar rápido permite:

- liberar recursos,
- informar adecuadamente,
- evitar colapsos en cascada.

Aunque el manejo concreto de timeouts depende de librerías externas, el concepto debe formar parte del diseño mental del programador desde etapas tempranas.

Logging profesional vs System.out.println

Imprimir mensajes por consola mediante System.out.println es una práctica aceptable únicamente en ejemplos académicos básicos. En sistemas reales, esta técnica es insuficiente y peligrosa.

El logging profesional permite:

- registrar niveles de severidad,
- conservar trazas históricas,
- separar información técnica de mensajes al usuario.

Aunque este libro no introduce una librería de logging específica, el principio es claro: los errores técnicos se registran, no se imprimen.

Capturar Exception: solo en los bordes del sistema

Capturar directamente Exception elimina toda información semántica del error y suele ser una señal de diseño deficiente. Esta práctica solo es aceptable en los bordes del sistema: controladores, puntos de entrada o capas de integración externa. En el núcleo del sistema, se deben capturar excepciones específicas, permitiendo un manejo preciso y evitando ocultar errores graves.

5.4. Modelo completo de gestión de errores en sistemas orientados a objetos

Después de analizar por separado los errores de entrada, de negocio y técnicos, resulta imprescindible integrar estos conceptos en un modelo coherente de trabajo. Un sistema profesional no puede tratar los errores de forma improvisada; necesita una estrategia clara que indique qué tipo de error es, cómo se expresa y dónde se gestiona. Este modelo no es una librería ni un framework, sino una metodología de diseño que guía las decisiones del programador.

Metodología propuesta

La metodología de gestión de errores se estructura en cinco pasos:

1. Clasificar el error

- ¿Es de entrada?
- ¿Es de negocio?
- ¿Es técnico?

2. Elegir el mecanismo adecuado

- Validación con resultado.
- Excepción de negocio.
- Excepción técnica.

3. Centralizar las reglas

- Validadores de entrada.
- Excepciones propias del dominio.
- Traducción técnica por capa.

4. Mantener la trazabilidad

- Nunca perder la causa original.
- Mensajes claros y contextualizados.

5. Gestionar en el nivel correcto

- UI: comunica.
- Servicio: decide.
- Infraestructura: informa.

Caso práctico integral: Sistema de pedidos con validación, negocio y fallos técnicos

En esta sección se desarrolla un caso práctico completo cuyo objetivo no es introducir nuevas técnicas, sino integrar de forma coherente todo lo aprendido en este capítulo. El propósito pedagógico es que el estudiante observe cómo conviven, en un mismo sistema, los errores de entrada, las reglas de negocio y los errores técnicos, y cómo cada uno se gestiona con un mecanismo distinto sin romper el diseño. El caso se ha diseñado deliberadamente para parecerse a situaciones reales que aparecen en sistemas profesionales, evitando ejemplos artificiales o excesivamente simplificados.

Contexto del problema: Un sistema permite crear pedidos, aplicar descuentos y confirmar el envío.

Contexto del dominio: El sistema permite gestionar pedidos de clientes. Un pedido:

- contiene una lista de productos,
- puede tener un descuento aplicado,
- pasa por distintos estados (CREADO, CONFIRMADO, ENVIADO),
- depende de un servicio externo de facturación.

Paso 1: Validación de entrada acumulada

A diferencia de ejemplos simplificados, en un sistema real no se detiene la validación en el primer error. Es preferible informar al usuario de todos los problemas detectados para evitar ciclos innecesarios de corrección.

```
public class ValidadorPedido {


    public ResultadoValidacion validarCreacion(List<String>
productos, Double descuento) {

        ResultadoValidacion resultado = ResultadoValidacion.
ok();


        if (productos == null || productos.isEmpty()) {

            resultado.agregarError("El pedido debe contener al
menos un producto". );

        }


        if (descuento == null) {

            resultado.agregarError("El descuento debe especi-
ficarse". );

        } else if (descuento < 0 || descuento > 50) {

            resultado.agregarError("El descuento debe estar
entre 0% y 50%". );

        }


        return resultado;

    }

}
```

Análisis;

- Se acumulan errores.

- No se lanzan excepciones.
- El flujo sigue siendo controlado.
- Se respeta la naturaleza de los errores de entrada.

Paso 2: Regla de negocio real (estado y coherencia)

Una vez superada la validación de entrada, el sistema debe imponer las reglas del dominio. Aquí aparece una regla clara: un pedido no puede confirmarse si ya fue enviado, y no puede confirmarse si no tiene productos válidos.

```
public class Pedido {

    private EstadoPedido estado = EstadoPedido.CREADO;

    public void confirmar() {
        if (estado == EstadoPedido.ENVIADO) {
            throw new ExcepcionNegocio("El pedido ya fue enviado y no puede confirmarse". );
        }
        estado = EstadoPedido.CONFIRMADO;
    }
}
```

Análisis

- El dato es válido.
- La operación es inválida.
- Se lanza excepción de negocio.
- El sistema protege su integridad.

Paso 3: Error técnico realista (servicio externo)

```
public class ServicioFacturacion {


    public void facturar(Pedido pedido) {
        try {
            // Simulación de fallo técnico
            throw new IOException("Servicio externo no disponible");
        } catch (IOException e) {
            throw new RuntimeException("Error técnico al facturar el pedido", e);
        }
    }
}
```

Análisis

- El error no depende del usuario.
- No es regla de negocio.
- Se preserva la causa.
- Se propaga con contexto.

Paso 4: Orquestación en el servicio de aplicación

```
public class PedidoService {


    private final ValidadorPedido validador = new ValidadorPedido();
    private final ServicioFacturacion facturacion = new Servi-
```

```
cioFacturacion();

    public ResultadoValidacion crearYConfirmarPedido(
            List<String> productos, Double descuento) {

        ResultadoValidacion r = validador.validarCreacion(pro-
ductos, descuento);
        if (!r.esValido()) return r;

        Pedido pedido = new Pedido();
        pedido.confirmar();

        facturacion.facturar(pedido);

        return ResultadoValidacion.ok();
    }
}
```

Paso 5: Manejo en la capa "controlador" (simulada)

```
public class PedidoController {

    public void procesarPedido(List<String> productos, Dou-
ble descuento) {

        PedidoService service = new PedidoService();
```

```
        try {
            ResultadoValidacion r = service.crearYConfirmarPedido(productos, descuento);
            if (!r.esValido()) {
                System.out.println("Errores de entrada:");
                r.getErrores().forEach(System.out::println);
                return;
            }
            System.out.println("Pedido procesado correctamente". );

        } catch (ExcepcionNegocio e) {
            System.out.println("Regla de negocio violada: " + e.getMessage());

        } catch (RuntimeException e) {
            System.out.println("Error técnico. Contacte a soporte". );
            e.printStackTrace(); // logging real en sistemas profesionales
        }
    }
}
```

Análisis final del caso

En este caso práctico se observa con claridad cómo cada tipo de error es tratado de acuerdo con su naturaleza, evitando soluciones improvisadas y manteniendo un diseño coherente.

Los errores de entrada se gestionan mediante objetos de resultado (ResultadoValidacion). Este enfoque permite detectar uno o varios problemas en los datos proporcionados sin interrumpir la ejecución normal del programa. El sistema permanece estable y devuelve información clara para que el usuario pueda corregir los errores detectados. En ningún momento se lanzan excepciones, ya que no existe una situación excepcional desde el punto de vista del sistema, sino datos inválidos que requieren corrección.

Las reglas de negocio, en cambio, se imponen mediante excepciones propias del dominio. Cuando una operación contradice una norma fundamental del sistema, por ejemplo, confirmar un pedido en un estado inválido, continuar el flujo sería incorrecto. En estos casos, la excepción detiene la ejecución de forma explícita y comunica claramente qué regla ha sido violada, protegiendo la integridad del modelo del dominio.

Los errores técnicos se manejan mediante propagación o traducción de excepciones, preservando siempre la causa original. Este tipo de errores no puede ser corregido por el usuario ni por la lógica de negocio, por lo que ocultarlos sería una mala práctica. Al propagar el error con contexto, el sistema permite un diagnóstico adecuado, registro de la falla y toma de decisiones en capas superiores.

Este enfoque demuestra que un diseño robusto no consiste en evitar los errores, sino en clasificarlos correctamente y aplicar el mecanismo adecuado en cada caso, logrando así sistemas más claros, mantenibles y profesionales.

Caso guiado completo por capas: Sistema de matrícula académica

En este apartado se desarrolla un caso guiado integral cuyo objetivo es consolidar de forma práctica y coherente todo lo estudiado en el Capítulo 5. A diferencia de ejemplos aislados, este caso muestra cómo conviven simultáneamente los distintos tipos de errores, entrada, negocio y técnicos, dentro de un sistema orientado a objetos estructurado por capas.

El dominio seleccionado es el de matrícula académica universitaria, por ser cercano al contexto del estudiante y representar un escenario realista en el que confluyen múltiples reglas, validaciones y dependencias técnicas. Este tipo de sistema es especialmente adecuado para ilustrar la importancia de un manejo correcto de errores, ya que una decisión incorrecta puede afectar registros académicos, cupos, prerrequisitos y la integridad institucional.

El propósito de este caso no es introducir nuevas técnicas, sino aplicar de manera sistemática los criterios ya estudiados, mostrando cómo un diseño bien pensado evita ambigüedades, duplicaciones y errores difíciles de diagnosticar.

Descripción general del dominio

El sistema permite a un estudiante realizar la matrícula en una asignatura durante un período académico. Para ello, intervienen los siguientes elementos conceptuales:

Datos de entrada proporcionados por la interfaz:

- código de la asignatura,
- paralelo seleccionado,
- número de créditos declarados.

Reglas de negocio propias del dominio académico:

- el estudiante debe cumplir los prerrequisitos,
- el paralelo debe tener cupos disponibles,
- no debe existir choque de horarios.

Dependencias técnicas del sistema:

- persistencia de la matrícula en base de datos,
- consulta a servicios internos del sistema académico.

Cada uno de estos aspectos da lugar a un tipo distinto de error, que no debe tratarse con el mismo mecanismo.

Paso 1: Validación de entrada (errores esperables y corregibles)

Antes de intentar cualquier operación académica, el sistema debe verificar que los datos recibidos cumplen condiciones mínimas de forma y consistencia básica. Estos errores no representan fallos del sistema, sino información incompleta o incorrecta.

Ejemplos de errores de entrada en este contexto:

- código de asignatura vacío,
- paralelo no seleccionado,
- número de créditos fuera de rango permitido.

Estas validaciones se gestionan mediante un objeto ResultadoValidacion, permitiendo acumular varios errores y devolverlos de forma conjunta.

```
public class ValidadorMatricula {

    public ResultadoValidacion validarEntrada(
        String codigoAsignatura, String paralelo, Integer
```

```
creditos) {

        ResultadoValidacion resultado = ResultadoValidacion.
ok();

        if (codigoAsignatura == null || codigoAsignatura.trim().
isEmpty()) {
            resultado.agregarError("El código de la asignatura
es obligatorio". );
        }

        if (paralelo == null || paralelo.trim().isEmpty()) {
            resultado.agregarError("Debe seleccionar un para-
lelo". );
        }

        if (creditos == null || creditos < 1 || creditos > 10) {
            resultado.agregarError("Los créditos deben estar
entre 1 y 10". );
        }

        return resultado;
    }
}
```

Análisis

- Se valida antes de modificar el sistema.
- No se lanzan excepciones.
- Se informa todo lo incorrecto en una sola respuesta.
- El flujo sigue siendo controlado.

Paso 2: Reglas de negocio (protección del dominio académico)

Una vez superada la validación de entrada, el sistema debe imponer las reglas que definen el dominio académico. En este punto, los datos pueden ser técnicamente válidos, pero la operación puede ser conceptualmente inválida.

Ejemplos de violaciones de negocio:

- el estudiante no cumple los prerrequisitos,
- el paralelo no tiene cupos disponibles,
- existe choque de horario con otra asignatura ya matriculada.

Estas situaciones no son corregibles por simple edición de datos, por lo que el sistema debe detener el proceso mediante excepciones de negocio.

```
public class MatriculaService {


    public void verificarReglasNegocio(Estudiante e, Asigna-
tura a) {


        if (!e.cumplePrerrequisitos(a)) {

            throw new PrerrequisitoNoCumplidoException(a.
getCodigo());
```

```
            }

            if (a.getCuposDisponibles() <= 0) {
                throw new CupoAgotadoException(a.getCodigo());
            }

            if (e.tieneChoqueHorario(a)) {
                throw new ExcepcionNegocio(
                        "Existe choque de horario con otra asignatura
matriculada".
                );
            }
        }
    }
```

Análisis

- El sistema impone reglas, no las negocia.
- La excepción comunica claramente la causa.
- El dominio queda protegido contra estados inválidos.
- El código refleja normas institucionales reales.

Paso 3: Error técnico (fallos de infraestructura inevitables)

Superadas las reglas de negocio, el sistema debe persistir la matrícula. En este punto pueden aparecer errores totalmente ajenos al usuario y al dominio, como fallos de base de datos o indisponibilidad del sistema académico.

```
public class MatriculaRepositorio {

    public void guardar(Matricula m) {
        try {
            // Simulación de fallo técnico
            throw new SQLException("Base de datos no disponible");
        } catch (SQLException e) {
            throw new ErrorPersistenciaException(
                "No se pudo guardar la matrícula", e
            );
        }
    }
}
```

Análisis

- El error no se corrige con validaciones.
- No se oculta.
- Se preserva la causa técnica.
- Se traduce a una excepción propia de infraestructura.

Paso 4: Orquestación del caso de uso en el servicio

El servicio integra las tres dimensiones del error: entrada, negocio y técnico, sin mezclarlas.

```
public class MatriculaAplicacionService {
```

```
    private final ValidadorMatricula validador = new Valida-
dorMatricula();

   private final MatriculaRepositorio repositorio = new Matri-
culaRepositorio();

    private final MatriculaService negocio = new Matricula-
Service();


   public ResultadoValidacion matricular(

         Estudiante e, Asignatura a,

            String codigoAsignatura, String paralelo, Integer
creditos) {


      ResultadoValidacion r =

        validador.validarEntrada(codigoAsignatura, paralelo,
creditos);

      if (!r.esValido()) return r;


      negocio.verificarReglasNegocio(e, a);


      Matricula m = new Matricula(e, a, paralelo, creditos);

      repositorio.guardar(m);


      return ResultadoValidacion.ok();

   }

}
```

Paso 5: Manejo en la capa de controlador (comunicar sin ocultar)

El controlador representa el borde del sistema y es responsable de comunicar correctamente lo ocurrido.

```
public class MatriculaController {

    public void procesarMatricula(...) {

        try {
            ResultadoValidacion r = servicio.matricular(...);

            if (!r.esValido()) {
                r.getErrores().forEach(System.out::println);
                return;
            }

            System.out.println("Matrícula realizada correctamente". );

        } catch (ExcepcionNegocio e) {
            System.out.println("No se pudo realizar la matrícula: " + e.getMessage());

        } catch (ErrorPersistenciaException e) {
            System.out.println("Sistema no disponible. Intente más tarde". );
```

```
            e.printStackTrace(); // logging técnico
        }
    }
}
```


Análisis

- El usuario recibe mensajes comprensibles.
- El detalle técnico no se expone.
- La causa se conserva para diagnóstico.
- El sistema no entra en estados incoherentes.

El caso de matrícula académica desarrollado en este epígrafe evidencia que la calidad de un sistema no depende únicamente de que "funcione", sino de cómo responde ante situaciones inválidas, restricciones del dominio y fallos inevitables del entorno técnico. A lo largo del ejemplo se ha mostrado que un mismo proceso puede enfrentarse a problemas de naturaleza muy distinta, y que tratarlos de forma uniforme conduce a diseños frágiles y confusos.

En primer lugar, se confirma que los errores de entrada forman parte del flujo normal del sistema y deben ser gestionados mediante validaciones explícitas que informen al usuario sin interrumpir la ejecución. El uso de un objeto de resultado permite acumular errores, mantener el control del flujo y evitar el uso innecesario de excepciones para situaciones esperables. Esta decisión simplifica el diseño y mejora la experiencia del usuario.

En segundo lugar, el caso deja claro que las reglas de negocio no se validan, se imponen. Cuando una operación contradice normas esenciales del dominio académico como prerrequisitos, cupos o conflictos de

horario, el sistema debe detener el proceso de forma clara y explícita mediante excepciones de negocio. De esta manera, el modelo del dominio se protege y el código expresa con precisión las reglas institucionales que gobiernan el sistema.

Finalmente, se observa que los errores técnicos requieren un tratamiento distinto, ya que no dependen del usuario ni del dominio. Su correcta traducción y propagación permite preservar la causa original del fallo, facilitar el diagnóstico y evitar que detalles de infraestructura contaminen capas superiores. El manejo adecuado en la capa de controlador demuestra que es posible comunicar errores de forma comprensible sin ocultar información crítica.

En conjunto, este caso integrador refuerza una idea central del capítulo: un diseño orientado a objetos robusto se construye clasificando correctamente los errores y aplicando el mecanismo adecuado en cada nivel del sistema. Al adoptar este enfoque, el programador deja de reaccionar de forma improvisada ante los fallos y comienza a diseñar software más claro, mantenible y alineado con la práctica profesional.

5.5. Checklist de revisión para autoevaluar validaciones y manejo de errores

Esta sección funciona como una lista de verificación práctica. Su propósito no es “memorizar reglas”, sino ayudarte a auditar tu propio diseño antes de darlo por terminado. La idea es que, al finalizar un ejercicio o proyecto, puedas recorrer este checklist y detectar los errores estructurales más comunes que hacen que un sistema sea frágil, confuso o difícil de mantener.

A diferencia de otros temas (donde un error produce un fallo inmediato), aquí muchas malas decisiones no fallan al instante: simplemente degradan el diseño, lo vuelven inconsistente, y con el tiempo aparecen bugs

difíciles de rastrear. Por eso este checklist es una herramienta de calidad.

Clasificación del error: ¿estoy reaccionando al problema correcto?

Pregunta 1: ¿Identifiqué si el error es de entrada, negocio o técnico antes de decidir la solución?

Buena señal: puedo decir "esto es entrada / esto es negocio / esto es técnico" en una frase clara.

Mala señal: todo lo trato con if o todo lo trato con try-catch, sin criterio.

Pregunta 2: Si el dato está mal formado (vacío, formato, rango), ¿lo traté como entrada y no como "excepción"?

Entrada es esperable: se valida y se devuelve feedback, no se "revienta" el flujo.

Pregunta 3: Si el dato es válido pero viola una regla del dominio, ¿lo traté como negocio y no como "entrada"?

Si es negocio, el sistema debe imponer la regla, no solo "decir que está mal".

Pregunta 4: Si el fallo es externo (archivo, BD, red), ¿lo traté como técnico y no como negocio?

Un error técnico no se resuelve con una validación; se propaga/traduce con causa.

5.6.2 Validaciones de entrada: calidad del diseño y coherencia

Pregunta 5: ¿Estoy validando la entrada antes de ejecutar lógica de negocio o persistencia?

Mala práctica típica: guardar primero y validar después ("validación tardía").

Pregunta 6: ¿Mis validaciones de entrada están centralizadas (ej. ValidadorEntrada) o dispersas por todo el sistema?

Si están dispersas: duplicación + reglas inconsistentes + mantenimiento costoso.

Pregunta 7: ¿Evité "arreglar" datos silenciosamente?

Ejemplos peligrosos:

- edad < 0 → edad = 0
- recortar IDs, rellenar con ceros, reemplazar texto inválido por default. Esto oculta el error y ensucia el sistema.

Pregunta 8: ¿Estoy diferenciando validación de forma vs validación de consistencia?

- Forma: null, vacío, regex, rango, tipo.
- Consistencia: reglas entre campos (fechaInicio < fechaFin, siFactura→RUC, etc.). Si no separas esto, la validación se vuelve confusa.

Pregunta 9: ¿Mi ResultadoValidacion acumula múltiples errores o solo devuelve uno?

En formularios reales, acumular es clave: permite reportar todo lo que falta sin ciclos de "corrige uno y vuelve".

Pregunta 10: ¿Estoy devolviendo mensajes útiles y accionables?

Mala señal: "Error", "Datos inválidos", "No se pudo".

Buena señal: "El RUC debe tener 13 dígitos numéricos" / "Debe seleccionar un paralelo".

Errores de negocio: excepciones propias y semántica del dominio

Pregunta 11: ¿Estoy usando excepciones de negocio para reglas del dominio (y no RuntimeException genérica)?

Si el error es del dominio, debe tener nombre del dominio:

Cupo Agotado Exception, Prerrequisito No Cumplido Exception, etc.

Pregunta 12: ¿La excepción expresa la regla violada con claridad y datos contextualizados?

Buena señal: incluye información útil: saldo actual, cupo, código de asignatura, estado del pedido, etc.

Mala señal: mensaje genérico sin contexto.

Pregunta 13: ¿Estoy evitando convertir errores de negocio en ResultadoValidacion?

Regla del dominio violada no es "campo mal escrito".

Si lo tratas como ResultadoValidacion, el flujo queda "suave" cuando debería detenerse.

Pregunta 14: ¿Estoy separando precondiciones técnicas del método vs reglas del negocio?

Ejemplo típico:

monto <= 0 → precondición / parámetro inválido (entrada o precondición técnica)

monto > saldo → negocio (regla del dominio)

Si no se explica ni se separa, el estudiante mezcla criterios y copia sin comprender.

Errores técnicos: propagación, trazabilidad y diagnóstico profesional

Pregunta 15: ¿Estoy preservando la causa original del error técnico?

Si capturo IOException o SQLException, ¿la incluyo como causa al relanzar?

throw new RuntimeException("..". , e);

Pregunta 16: ¿Evité capturar Exception o Throwable salvo en el borde del sistema?

Capturar todo produce “bugs fantasma”: se pierde el origen real.

Pregunta 17: ¿Estoy usando try-with-resources cuando manejo recursos (archivos, streams)?

Si uso finally manual, reviso si realmente es necesario o si es código frágil.

Pregunta 18: ¿Hice traducción por capa cuando corresponde?

Repositorio: traduce fallos de BD a error de persistencia.

Servicio: decide si propaga o encapsula.

Controlador: muestra mensaje amigable y registra detalle técnico.

Pregunta 19: ¿Estoy usando logging apropiado en vez de System.out.println para errores reales?

System.out no sirve para auditoría ni diagnóstico serio.

Un sistema profesional registra contexto: operación, usuario, hora, causa.

Pregunta 20: ¿Estoy evitando reintentos automáticos sin criterio?

Reintentar indiscriminadamente puede duplicar operaciones y empeorar fallos.

Un retry solo se justifica si:

- es idempotente,
- tiene límite,
- tiene espera/backoff,
- y el fallo es transitorio.

Señales de alerta de mal diseño (diagnóstico rápido)

- Hay try { ... } catch (Exception e) { } (vacío o mensaje genérico).
- Existe la misma validación copiada en 3 clases distintas.
- Hay reglas del negocio expresadas como System. out.println("Error").
- Hay datos "normalizados" sin avisar al usuario.
- Se capturan excepciones técnicas y se devuelven como "Dato inválido".
- Hay un ValidadorEntrada que ya hace negocio ("no puede retirar más de saldo").
- Hay controladores que contienen lógica de negocio y validaciones mezcladas.

Si aparecen 2 o más de listado anterior, el diseño requiere refactorización.

Con este checklist, el estudiante dispone de un criterio práctico para revisar su propio código con estándares profesionales: no basta con que el programa funcione, debe ser coherente en cómo valida, cómo impone reglas y cómo comunica fallos técnicos. En el siguiente epígrafe se realizará el cierre del capítulo, sintetizando los aprendizajes y estableciendo la continuidad hacia el próximo tema del libro, donde estas decisiones se conectan con la organización y calidad del software a mayor escala.

En este capítulo se ha construido una visión integral y sistemática sobre la validación y el manejo de errores en sistemas orientados a objetos, superando el enfoque limitado que reduce estos temas al uso aislado de condicionales o bloques try-catch. A lo largo de los epígrafes se demostró que los errores no

son todos iguales y que su correcta gestión comienza, necesariamente, por una clasificación consciente: errores de entrada, errores de regla de negocio y errores técnicos. Esta distinción no es un formalismo académico, sino la base para tomar decisiones de diseño coherentes y sostenibles.

Se profundizó en el tratamiento de los errores de entrada, mostrando que forman parte del flujo normal del sistema y deben gestionarse mediante validaciones explícitas y centralizadas, apoyadas en objetos de resultado que permitan acumular fallos y comunicar al usuario información clara y accionable. De igual manera, se estableció que las reglas de negocio no se validan, se imponen, y que las excepciones propias del dominio constituyen el mecanismo adecuado para expresar y hacer cumplir estas normas, protegiendo la integridad conceptual del sistema. Por su parte, los errores técnicos fueron analizados como información crítica que nunca debe ocultarse, sino propagarse o traducirse de forma controlada, preservando siempre la causa original para facilitar el diagnóstico y el mantenimiento.

El capítulo también introdujo una metodología unificada de gestión de errores, integrando los distintos mecanismos estudiados y mostrando cómo se aplican de manera coordinada en sistemas reales. Los casos guiados y el checklist de revisión permitieron trasladar la teoría a la práctica, ofreciendo al estudiante herramientas concretas para auditar sus propios diseños, identificar anti-patrones y corregir decisiones que, aunque no siempre generan fallos inmediatos, deterioran progresivamente la calidad del software.

En conjunto, este capítulo consolida una competencia fundamental del desarrollo profesional: saber reaccionar ante situaciones inválidas con criterio de diseño, y no de forma improvisada. Validar correctamente, imponer

reglas con claridad y manejar fallos técnicos con responsabilidad transforma programas frágiles en sistemas robustos, comprensibles y mantenibles.

Epílogo

La programación orientada a objetos y el diseño de software no constituyen únicamente un conjunto de técnicas para escribir código, sino una forma de pensar y estructurar soluciones a problemas complejos. A lo largo de este libro se ha demostrado que comprender conceptos como abstracción, encapsulamiento, relaciones entre clases, principios SOLID y modelado UML permite al estudiante desarrollar sistemas más claros, mantenibles y coherentes con el dominio del problema.

El recorrido propuesto en los distintos capítulos ha buscado intencionalmente ir más allá de la sintaxis de un lenguaje de programación. El énfasis ha estado puesto en el razonamiento de diseño, en la toma de decisiones técnicas fundamentadas y en la identificación de buenas y malas prácticas que impactan directamente en la calidad del software. Esta aproximación resulta especialmente relevante en contextos académicos donde el aprendizaje debe trascender la resolución puntual de ejercicios y proyectarse hacia escenarios reales de desarrollo e investigación.

En el ámbito universitario actual, los estudiantes de Ingeniería en Sistemas, Tecnología en Big Data e Inteligencia de Negocios y Tecnología en Desarrollo de Software se enfrentan a desafíos que requieren no solo programar, sino diseñar soluciones robustas, escalables y comprensibles para equipos multidisciplinarios. La correcta estructuración del software, la gestión adecuada de dependencias, el uso consciente de herencia, composición y polimorfismo, así como el dominio del modelado UML, se convierten en competencias transversales indispensables.

Este libro se concibe, por tanto, como un material de estudio que acompaña al estudiante a lo largo de su

formación académica y que sirve de base para su participación en proyectos de investigación aplicada. En particular, los fundamentos aquí abordados resultan esenciales para integrarse de manera efectiva en iniciativas institucionales vinculadas a la inteligencia artificial, el aprendizaje profundo, el análisis de datos y el desarrollo de sistemas complejos, donde la calidad del diseño de software es un requisito crítico.

Asimismo, el texto pretende fomentar una actitud reflexiva frente al desarrollo de software. Comprender por qué una solución es correcta, por qué otra es frágil o por qué una clase se convierte en una "clase Dios" es parte del proceso de maduración profesional del estudiante. Esta capacidad de análisis crítico es la que distingue a un programador que solo implementa instrucciones de aquel que diseña soluciones sostenibles y alineadas con estándares académicos e industriales.

Finalmente, este libro no debe entenderse como un punto de llegada, sino como un punto de partida. Los conceptos aquí desarrollados constituyen la base sobre la cual se construyen arquitecturas más avanzadas, patrones de diseño, pruebas de software, sistemas distribuidos y aplicaciones basadas en inteligencia artificial. Se espera que el lector utilice este material como referencia constante, regrese a él para justificar decisiones de diseño y lo integre activamente en su proceso de aprendizaje, investigación y desarrollo profesional.

Referencias

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. do N., & Dig, D. (2020). *30 years of software refactoring research: A systematic literature review* (arXiv:2007.02194). arXiv. https://doi.org/10.48550/arXiv.2007.02194

Akhtar, S. M., Nazir, M., Ali, A., Khan, A. S., & Atif, M. (2022). *A systematic literature review on software-refactoring techniques, challenges, and practices*. Research Square. https://doi.org/10.21203/rs.3.rs-1472519/v1

Alemán Flores, M. (2025). Discrimination criteria for modeling association, aggregation, and composition in UML class diagrams. En A. Quesada Arencibia, M. Affenzeller, & R. Moreno Díaz (Eds.), *Computer aided systems theory – EUROCAST 2024* (pp. 443–457). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-82949-9_39

Al-Fedaghi, S. (2021). *Classes in object-oriented modeling (UML): Further understanding and abstraction*. https://doi.org/10.22937/IJCSNS.2021.21.5.21

Al-Fedaghi, S. (2022). *Conceptual modeling of aggregation: An exploration* (arXiv:2208.11171). arXiv. https://doi.org/10.48550/arXiv.2208.11171

Cabral, R., Kalinowski, M., Baldassarre, M. T., Villamizar, H., Escovedo, T., & Lopes, H. (2024). Investigating the impact of SOLID design principles on machine learning code understanding. *3rd International Conference on AI Engineering - Software Engineering for AI*. New York, United States.

Chaudhary, P., Agrawal, L., & Ali, A. (2025). Modern programming languages—Characteristics and recommendations for instruction. *Issues in Information Systems, 26*(2), 281–291. https://doi.org/10.48009/2_iis_122

Chávez-Cárdenas, M. d. C., Fernández-Marín, M. Á., & Lamí-Rodríguez del Rey, L. E. (2025). *Educational Web and Artificial Intelligence: Transforming Contemporary Learning*. Sophia Editions.

Chaw, M. (2024). *Object-oriented development and inheritance: A comprehensive study*. https://doi.org/10.13140/RG.2.2.34643.44323

Chen, Y., & Huang, L. (2025). *MATLAB roadmap to applications: Volume I fundamental*. Springer Nature.

Fernández Marín, M. Á., & González Tolmo, D. (2020). Propuesta de fusión de una metodología para multimedia con el Proceso Unificado evidenciado en un caso real. Revista Metropolitana De Ciencias Aplicadas, 3(3), 133-140. https://doi.org/10.62452/bxqdzf25

Fernández Marín, M. Á., Alfonso Moreira, Y., Valladares González, M. G., & Alfonso García, A. B. (2022). Articulación de la academia, la investigación y vinculación: Concepciones y proyecciones desde la práctica virtual. *Revista Universidad y Sociedad, 14*(4), 505–512. http://scielo.sld.cu/pdf/rus/v14n4/2218-3620-rus-14-04-505.pdf

Fernández Marín, M. Á., Nacimba Quinga, A. C., Gutiérrez Rodríguez, F. Á., & González Tolmo, D. (2019). Multimedia educativa para el desarrollo de habilidades lógico-matemáticas en niños de inicial II. *Revista Metropolitana de Ciencias Aplicadas, 2*(2), 204–213. https://doi.org/10.62452/3g985n27

Fernández Marín, M. Á., Valladares González, M. G., & Alfonso Moreira, Y. (2022). Propuesta interactiva para el desarrollo de las competencias digitales. *Revista Metropolitana de Ciencias Aplicadas, 5*(2), 89–95. https://doi.org/10.62452/3mb4rk57

Fernández-Marín, M. Á., Montano-Rodríguez, F., González-Tolmo, D., & Manso-Rivero, Y. (2024). Interdisciplinariedad entre la materia de Sistemas de Gestión de Bases de Datos e inteligencia artificial de Universidad Metropolitana del Ecuador. *Revista Mexicana de Investigación e Intervención Educativa, 3*(2), 81–88. https://doi.org/10.62697/rmiie.v3i2.87

Ghorpade, S., & Wadkar, V. (2024). Exploring the concepts of object-oriented programming: Objects, classes, inheritance, polymorphism, abstraction, and encapsulation. *International Journal of Novel Research and Development, 9*(9), 412–414. https://ijnrd.org/viewpaperforall.php?paper=IJNRD2409250

Haroon, E., Wassif, K. T., & Zaid, L. A. (2025). From code to insight: Studying code representation techniques for ML-based God class detection to support intelligent IDEs. *Automated Software Engineering, 32*(2), 66. https://doi.org/10.1007/s10515-025-00534-4

Heričko, T., & Šumak, B. (2023). Exploring maintainability index variants for software maintainability measurement in object-oriented systems. *Applied Sciences, 13*(5), 2972. https://doi.org/10.3390/app13052972

Khaleel, S. I., & Mahmood, R. A. (2024). Automatic software refactoring to enhance quality: A review. *Jiteki, 10*(4). https://eprints.uad.ac.id/79755/1/6-Automatic%20Software%20Refactoring%20to%20Enhance%20Quality%20A%20Review.pdf

Lavand, M., & Pawar, R. (2025). A study on abstraction in object oriented programming with Java and its implementation in developing programs. *IJSAT - International Journal on Science and Technology, 16*(2). https://doi.org/10.71097/IJSAT.v16.i2.3438

Magableh, A. A., Bani Ata, H., Saifan, A. A., & Rawashdeh, A. (2024). Towards improving aspect-oriented software reusability estimation. *Scientific Reports, 14*, 13214. https://doi.org/10.1038/s41598-024-62995-z

Meyer, B. (1994). *Object oriented software construction*. Prentice-Hall.

Parmar, M. D., & Parmar, S. (2024). Survey on concept of object-oriented programming. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 10*(2), 427–431. https://doi.org/10.32628/CSEIT243647

Priolo, S. (2009). *Métodos ágiles*. Creative Andina Corp.

Ramírez Jiménez, M. del R., Pulido Hernández, K., Rivera Orozco, C. E., Gómez Torres, N. A., Serrano Zúñiga, L., & Orozco Torres, L. M. O. (2024). UML: Una manera de representar, interpretar, analizar y desarrollar el pensamiento computacional. *RIDE Revista Iberoamericana para la Investigación y el Desarrollo Educativo,15*(29). https://doi.org/10.23913/ride.v15i29.2196

Rumbaugh, J., Jacobson, I., & Booch, G. (2006). *El lenguaje unificado de modelado. Manual de referencia*. Pearson Education.

Sánchez Durán, K. D. (2026). Object-oriented programming: Foundations, evolution, and industrial applications. *Asian Journal of Science, Technology, Engineering, and Art, 4*(1), 61–67. https://doi.org/10.58578/ajstea.v4i1.8093

Shah, K. P., & Grant, E. S. (2024). Verification and validation of software system class models. *8th International Conference on Information System and Data Mining.* Los Angeles, USA.

Shah, K., & Grant, E. (2022). Towards simplifying and formalizing UML class diagram generalization/ specialization relationship with mathematical set theory. *6th International Conference on Information System and Data Mining.* Silicon Valley, USA.

Sommerville, I. (2020). *Engineering software products: An introduction to modern software engineering*. Pearson.

Vargas Ortega, G. A. (2021). Lineamientos para el diseño de aplicaciones web soportados en patrones GRASP. *Ciencia e Ingeniería: Revista de investigación interdisciplinar en biodiversidad y desarrollo sostenible, ciencia, tecnología e innovación y procesos productivos industriales, 8*(2). https://dialnet.unirioja.es/descarga/articulo/8742482.pdf

Vera Vera, J. B., & Vera Vera, J. R. (2023). El papel de la programación orientada a objetos en el desarrollo de software sostenible y escalable. *Universidad Ciencia y Tecnología, 27*(121), 85–94. https://doi.org/10.47460/uct.v27i121.757

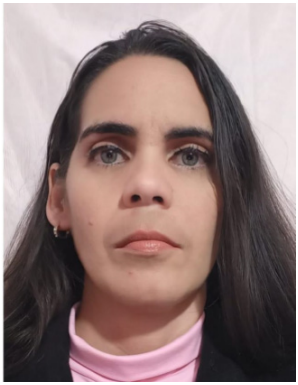
Vincent Gumonan, K. M., Sta Romana, C. L., & Pantaleon, C. (2024). Object-oriented programming teaching methods and instructional strategy in selected higher education institutions in the Philippines. *8th International Conference on Digital Technology in Education.* Berlin, Germany.

Miguel Ángel Fernández Marín

Profesional con dieciocho años de experiencia en ciencias informáticas y en la enseñanza universitaria de matemáticas e informática. Su trayectoria como especialista en informática se ha centrado en el desarrollo de sistemas empresariales, análisis, diseño y automatización de procesos de negocio; así como en la gestión de la calidad del software, desarrollada en la Universidad de las Ciencias Informáticas de Cuba. Durante su carrera docente universitaria, alcanzó la categoría de profesor asistente otorgada por el Ministerio de Educación Superior de Cuba y obtuvo el grado científico de Máster en Bioinformática y Biología Computacional. En la carrera de Informática de la Universidad de las Ciencias Informáticas de Cuba, impartió asignaturas como Matemática 1, 2 y 4; Matemática Discreta 1 y 2; Álgebra Lineal; y Modelos y Simulación. Además, en la carrera de Telecomunicaciones del Instituto Superior "ISUTIC" en Luanda, Angola, enseñó Álgebra, Análisis Vectorial y Análisis Matemático de Sistemas y Señales. En la Universidad Metropolitana del Ecuador ha impartido cursos como Modelos y Simulación, Data Warehouse, Herramientas Informáticas, Gerencia de Sistemas,

Ingeniería de Software, Análisis y Diseño de Sistemas Informáticos, Ingeniería de Software 1 y 2, Sistemas Operativos 1, Comunicaciones Inalámbricas, Calidad de Software, Curso Propedéutico de Razonamiento Lógico, Análisis Matemático II, Cálculo Diferencial e Integral, Estadística Inferencial, Programación 1, Estructuras de Datos y Sistemas de Gestión de Bases de Datos 1 y 2. Su desempeño profesional incluye resultados destacados en investigación, evidenciados por artículos publicados, participación en eventos académicos y la publicación de un libro.

ORCID: https://orcid.org/0000-0002-6132-539X



Débora González Tolmo

Ingeniera en Ciencias Informáticas graduada en 2009 en la Universidad de las Ciencias Informáticas(UCI) en Cuba. Con una trayectoria que supera los 13 años de experiencia, ha logrado consolidarse en el ámbito del aseguramiento de la calidad (QA/QC), la automatización de pruebas y la ingeniería de software. Su carrera profesional en sectores críticos como el financiero y el de servicios tecnológicos con una profunda vocación por la docencia universitaria y la investigación científica. En el sector corporativo,

ha participado en proyectos de alta complejidad en Ecuador, destacándose actualmente como Analista QC y Automatización en Netby para Produbanco, donde se especializa en el diseño de planes de prueba bajo estándares ISTQB, en la ejecución de pruebas de rendimiento con herramientas de vanguardia como VUGen y LoadRunner y automatización de prueba funcionales. Su pericia técnica abarca un amplio espectro de tecnologías, desde la automatización con Selenium y Cypress hasta la gestión de bases de datos en entornos Oracle, PostgreSQL y SQL Server, así como el análisis de APIs mediante Postman y JMeter. Ha trabajado en empresas como OpcionAuto y APPLIOS, donde gestionó la certificación de sistemas bancarios y la automatización de servicios web. Su faceta académica es igualmente prolífica. Durante casi una década en la UCI, ejerció como profesora de asignaturas como Arquitectura de Software e Ingeniería de Software, mientras desempeñaba roles de analista principal en proyectos de impacto social como el Sistema Informatizado de Urgencias Médicas (SIUM) y el proyecto Teleconsulta. Esta pasión por el conocimiento la llevó a obtener diplomados en Informática Médica y Software Educativo, además de completar cursos internacionales en la Universidad de Minnesota y Georgia Tech. Como investigadora, ha contribuido significativamente al acervo científico con múltiples publicaciones en revistas especializadas como Conrado y la Revista Metropolitana de Ciencias Aplicadas. Sus trabajos abordan desde el desarrollo de habilidades informacionales y lógica-matemática en niños, hasta la fusión de metodologías para multimedia y estadística inferencial.

ORCID: https://orcid.org/0000-0002-8890-130X

Fundamentos del Diseño Orientado a Objetos en Ingeniería de Software: Desarrollo en Java como base para la formación investigativa es una guía integral que combina teoría, práctica y reflexión crítica sobre la Programación Orientada a Objetos. Este libro ofrece a estudiantes y profesionales las herramientas necesarias para comprender, diseñar y evaluar software robusto, mantenible y escalable, utilizando el lenguaje de programación Java como referencia. Desde los conceptos fundamentales de clases, objetos y relaciones, hasta principios avanzados como responsabilidad única, polimorfismo, herencia múltiple y genericidad, el texto proporciona explicaciones claras, ejemplos prácticos, diagramas del Lenguaje Unificado de Modelado y ejercicios integradores que fomentan un aprendizaje activo y profundo. Su enfoque en diseño extensible, patrones de diseño y manejo profesional de errores prepara al lector para enfrentar sistemas complejos y cambiantes, garantizando soluciones eficientes y sostenibles. Además, el libro promueve la formación investigativa, estimulando la reflexión crítica, el análisis metodológico y la capacidad de tomar decisiones de diseño fundamentadas. Está pensado como material de apoyo académico y herramienta de referencia profesional, fortaleciendo competencias clave para el desarrollo de software en entornos educativos, de investigación y proyectos de innovación tecnológica. Con esta obra, se invita al lector a explorar la Programación Orientada a Objetos no solo como un conjunto de técnicas, sino como un enfoque disciplinado, creativo y estratégico para construir software de calidad y enfrentar los retos del desarrollo moderno.

Editorial UMET

ISBN: 978-9942-7448-3-8