Tema 6: Introducción al Diseño

BLOQUE III: DISEÑO DE LOS SISTEMAS SOFTWARE

Ingeniería del Software
Grado en Ingeniería Informática
Curso 2024/2025

David Cáceres Gómez

UNIVERSIDAD
DE
CÓRDOBA

Índice

1. Introducción

El diseño en ingeniería del software se basa en principios, conceptos y buenas prácticas orientadas al desarrollo de productos de software de calidad. Es fundamental porque conecta "el mundo de los humanos" con "el mundo de la tecnología", siendo una etapa donde la creatividad, el juicio, la experiencia y la práctica de los ingenieros resultan cruciales.

Este proceso implica la búsqueda y evaluación de alternativas para converger hacia una solución que satisfaga los requisitos de la mejor manera posible. Aunque existen metodologías de diseño, lo esencial es comprender los principios fundamentales, ya que el diseño se ubica en el núcleo tecnológico de la ingeniería del software y es independiente del modelo de proceso utilizado.

Tras analizar y modelar los requisitos, el diseño es la etapa previa a la construcción (código y pruebas) y organiza la información necesaria en cuatro niveles clave, proporcionando las bases para un desarrollo exitoso.

En la Figura 1 se ilustra el flujo de información en el proceso de diseño de software, donde el modelo de requerimientos, compuesto por elementos basados en escenarios, clases, flujo y comportamiento, sirve como base principal para las tareas de diseño.

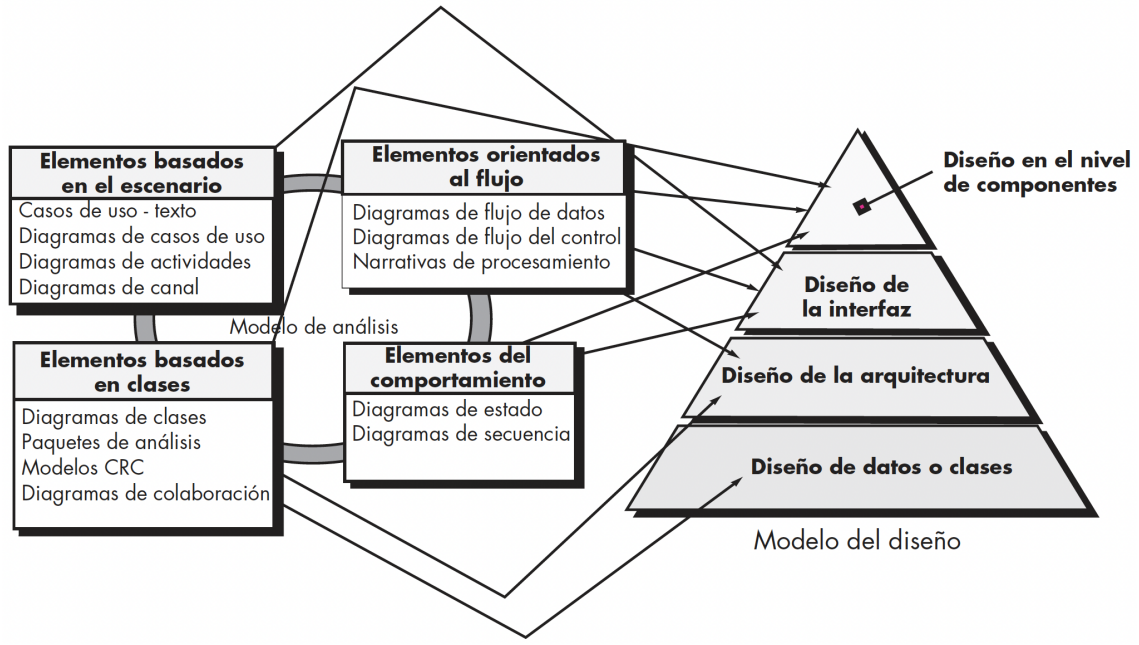


Figura 1: Traducción del modelo de requerimientos al modelo de diseño [1]

2. Conceptos de Diseño

A lo largo de la historia de la ingeniería del software, varios conceptos fundamentales han evolucionado y perdurado, sirviendo como base para aplicar métodos de diseño más avanzados. Estos principios permiten dividir el software en componentes, separar detalles técnicos de representaciones conceptuales y establecer criterios de calidad, ayudando a los desarrolladores a crear soluciones adecuadas en lugar de limitarse a producir programas funcionales obsoletos. Pressman [1] definió los siguientes conceptos:

- **Abstracción:** Consiste en trabajar en distintos niveles de detalle para resolver un problema. A niveles bajos, se introducen más elementos específicos de la implementación, combinando la terminología del problema con la técnica. Aplica tanto a datos como a procesos.
- **Arquitectura:** Define la estructura del software en unidades más pequeñas y sus interrelaciones. Incluye patrones arquitectónicos para problemas recurrentes y utiliza modelos y lenguajes propios (ADL).
- **Patrones:** Soluciones prácticas y reutilizables a problemas comunes de diseño, adaptadas a contextos específicos.
- **Separación de intereses (*separation of concerns*):** Divide un problema en partes independientes para facilitar su resolución. Este enfoque reduce tiempo y esfuerzo, pero debe equilibrarse para evitar complejidades excesivas al integrar las partes.
- **Modularidad:** Derivada de la separación de intereses, permite que el software sea manejable intelectualmente. Previene diseños monolíticos, facilitando la comprensión y el mantenimiento.
- **Ocultación de la información:** Diseña módulos de forma que mantengan sus detalles internos privados, limitando el acceso a datos y algoritmos solo dentro de cada módulo.
- **Independencia funcional:** Cada módulo realiza tareas específicas y ofrece una interfaz simple, promoviendo la reusabilidad y facilitando las pruebas al reducir errores propagados. Se evalúa mediante cohesión y acoplamiento.
- **Refinamiento:** Proceso iterativo de diseño de arriba hacia abajo, que descompone funciones abstractas en elementos concretos. Complementa la abstracción al revelar detalles progresivamente.
- **Refactorización:** Reorganización del código sin alterar su comportamiento observable, mejorando su comprensión y reduciendo costos de modificación.

La **cohesión** y el **acoplamiento** son dos conceptos fundamentales en el diseño de software. Ambos se refieren a la calidad de la estructura interna y externa de los módulos de un sistema y son claves para crear software fácil de mantener, entender y escalar.

Cohesión

La cohesión mide qué tan bien las responsabilidades y funciones dentro de un módulo están relacionadas entre sí. En otras palabras, evalúa el grado en que los elementos de un módulo trabajan juntos para lograr un único propósito.

- **Alta cohesión:** Un módulo tiene alta cohesión cuando todas sus funciones están estrechamente relacionadas y colaboran para cumplir una tarea específica. Esto facilita el mantenimiento y la reutilización del módulo.

  Ejemplo: Una clase 'Factura' que maneja todas las operaciones relacionadas con facturas, como calcular el total, aplicar descuentos y generar el recibo.

- **Baja cohesión:** Un módulo tiene baja cohesión cuando sus funciones son poco relacionadas o abarcan múltiples propósitos. Esto puede hacer que sea difícil de entender y mantener.

  Ejemplo: Una clase 'Utilidades' que incluye funciones diversas como calcular intereses, validar correos electrónicos y generar reportes.

Acoplamiento

El acoplamiento mide el grado de interdependencia entre diferentes módulos de un sistema. Indica cuánto afecta un cambio en un módulo a otros módulos.

- **Bajo acoplamiento:** Los módulos están diseñados de manera que su interacción sea mínima y se comuniquen a través de interfaces bien definidas. Esto reduce la posibilidad de que un cambio en un módulo afecte a otros.

  Ejemplo: Un módulo 'ProcesamientoDePago' que se comunica con un módulo 'Notificaciones' mediante una interfaz estándar.

- **Alto acoplamiento:** Los módulos están estrechamente interconectados y dependen directamente unos de otros. Esto hace que los cambios sean riesgosos, ya que pueden romper la funcionalidad en otros módulos.

  Ejemplo: Un módulo 'Inventario' que directamente accede y modifica datos en un módulo 'Clientes'.

Relación entre cohesión y acoplamiento

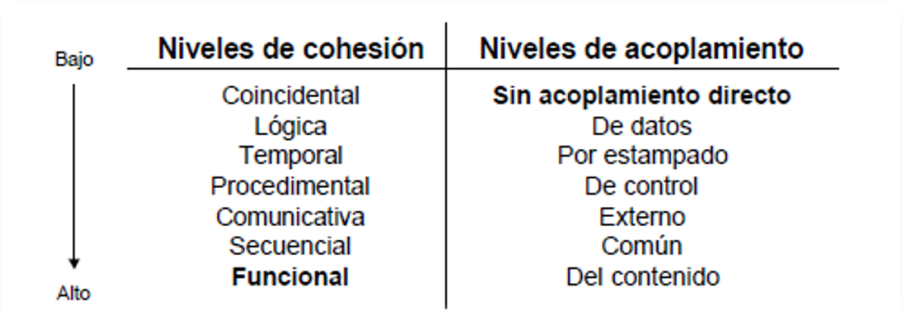


Figura 2: Niveles de cohesión y acoplamiento [2]

Lo ideal en un diseño es buscar **alta cohesión** (módulos enfocados en tareas específicas) y **bajo acoplamiento** (módulos independientes con interacciones mínimas).

Por el contrario pueden aparecer varios problemas comunes en el diseño de software:

- **Alta cohesión + Alto acoplamiento:** Los módulos están bien definidos, pero están demasiado interconectados.
- **Baja cohesión + Bajo acoplamiento:** Los módulos son independientes pero tienen responsabilidades dispersas o poco relacionadas, lo que dificulta su comprensión y reutilización.

Ejemplo

En un sistema de gestión de pedidos:

- **Alta cohesión:** Una clase 'Pedido' se enfoca exclusivamente en funciones relacionadas con los pedidos, como agregar productos, calcular el total y confirmar el pedido. Esto asegura que la clase tenga una única responsabilidad clara y bien definida.
- **Bajo acoplamiento:** La clase 'Pedido' interactúa con la clase 'Notificaciones' a través de una interfaz. Esto permite cambiar la implementación de 'Notificaciones' sin afectar el funcionamiento de la clase 'Pedido', facilitando la adaptabilidad y el mantenimiento del sistema.

3. Principios Básicos de Diseño

Existen varios principios de diseño software para hacer código de calidad como por ejemplo:

- *Don't Repeat Yourself* (**DRY**), evitar la duplicación de código.
- *You ain't gonna need it* (**YAGNI**), no lo hagas hasta que no lo necesites.
- *Keep it simple, stupid* (**KISS**), mantenlo simple, estúpido.
- **SOLID**

Principios SOLID

Los principios SOLID, propuestos inicialmente por Robert C. Martin y popularizados por Michael Feathers, son cinco reglas fundamentales para el diseño de clases en programación orientada a objetos [3]. Su objetivo es promover un código limpio, comprensible, mantenible y colaborativo. Estos principios ayudan a entender la importancia de patrones de diseño y arquitectura de software. Representados por el acrónimo SOLID, incluyen:

- El Principio de responsabilidad única (**S**ingle Responsibility Principle): Una clase debe tener una única razón para cambiar, es decir, debe encargarse de una sola tarea o responsabilidad dentro del sistema.
- El Principio Abierto-Cerrado (**O**pen-Closed Principle): Las clases deben estar abiertas a la extensión pero cerradas a la modificación, lo que significa que se deben poder añadir funcionalidades sin cambiar el código existente.
- El Principio de sustitución de Liskov (**L**iskov Substitution Principle): Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento esperado del programa.
- El Principio de segregación de interfaz (**I**nterface Segregation Principle): Una clase no debería verse obligada a implementar interfaces que no utiliza. Es preferible dividir las interfaces grandes en otras más específicas.
- El Principio de inversión de dependencia (**D**ependency Inversion Principle): Las clases de alto nivel no deben depender de clases de bajo nivel, sino de abstracciones. Esto promueve sistemas flexibles y menos acoplados.

Su aplicación mejora la calidad del software y la colaboración entre desarrolladores.

4. Niveles de Diseño

Según la pirámide de la Figura 1, el modelado del diseño se organiza en distintos niveles que guían el proceso de transformación de los requisitos en una solución técnica integral.

4.1. Diseño de Datos

El diseño de datos se centra en la organización, almacenamiento y acceso a los datos que utiliza el sistema. Es esencial para garantizar que los datos sean consistentes, íntegros y fáciles de manejar. Tiene una serie de elementos claves:

- Especificación de estructuras de datos: Identificación de archivos, tablas, relaciones y claves primarias/foráneas.
- Normalización de datos: Elimina redundancias y asegura consistencia.
- Selección de estructuras eficientes: Arrays, listas, árboles, grafos, etc.

Un diseño de datos eficiente mejora el rendimiento del sistema, reduce errores en el manejo de la información y facilita futuras ampliaciones.

Ejemplo:

Para un sistema de gestión de biblioteca:

- Tablas: 'Libros', 'Usuarios', 'Préstamos'.
- Relación: Cada préstamo está asociado a un usuario y un libro específico.

4.2. Diseño Arquitectónico

Define la estructura general del sistema en términos de módulos y sus interacciones. Se utiliza para organizar el software en componentes que sean manejables y escalables. Sus principales elementos son:

- Identificación de módulos principales: Dividir el sistema en componentes como interfaz, lógica de negocio y datos.

- Patrones arquitectónicos: MVC (Modelo-Vista-Controlador), microservicios, cliente-servidor.
- Diagramas arquitectónicos: Representaciones gráficas del diseño general.

Proporciona una visión global del sistema y asegura que los módulos sean independientes y fácilmente integrables.

Ejemplo:

Un sistema de comercio electrónico:

- Módulos: 'Gestión de Productos', 'Carrito de Compras', 'Gestión de Pedidos'.
- Patrón: Arquitectura en capas (capa de presentación, lógica de negocio, capa de datos).

4.3. Diseño de Interfaz

Se centra en cómo los módulos del sistema interactúan entre sí, con otros sistemas externos y con los usuarios. Este diseño contiene:

- Diseño de interfaces de usuario (UI): Define la interacción entre operadores y el sistema.
- Interfaces de programación (API): Especifica métodos y protocolos para que los módulos se comuniquen.
- Interacción con sistemas externos: Define los estándares y protocolos (REST, SOAP, etc.).

Un diseño de interfaz bien pensado garantiza la usabilidad, la interoperabilidad y la eficiencia.

Ejemplo:

Un sistema bancario:

- UI: Panel de usuario para consultar saldo y realizar transferencias.
- API: Una API que permite que cajeros automáticos consulten datos de la cuenta en tiempo real.

4.4. Diseño Procedimental o de las Funciones

Define los algoritmos y procesos internos necesarios para que el software cumpla con los requisitos funcionales. Tiene los siguientes elementos clave:

- Diseño de algoritmos: Instrucciones paso a paso para realizar tareas específicas.
- Diagrama de flujo: Representación gráfica del flujo lógico de un procedimiento.
- Pseudocódigo: Describe los pasos de forma textual y estructurada.

Proporciona una base clara para implementar las funcionalidades de cada módulo, asegurando precisión y eficiencia.

Ejemplo:

Un algoritmo para calcular el precio final de un pedido:

1. Sumar los precios de los productos.
2. Aplicar descuentos según las promociones.
3. Añadir impuestos.

5. Patrones de Diseño

Los **patrones de diseño software** (*software design patterns*) son soluciones generables y reutilizables a problemas comunes en el diseño de software.

Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su **efectividad** resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser **reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Por tanto, **un patrón no es una solución en si**, sino una plantilla o descripción para desarrollar una solución.

Los patrones de diseño se clasifican en tres categorías principales: creacionales, estructurales y de comportamiento. La Tabla 1 presenta estos tipos junto con los patrones correspondientes a cada categoría.

Creacionales	Estructurales	Comportamiento
Factory Method	Adapter	Chain of Responsability
Abstract Factory	Bridge	Command
Builder	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
Model View Controller (MVC)	Flyweight	Memento
	Proxy	Observer
	Module	State
		Strategy
		Template Method
		Visitor

Tabla 1: Clasificación de los patrones de diseño software

A continuación se verán varios ejemplos dentro de cada tipo de patrón, sacados del libro original del GoF (*Gang Of Four*) [4].

5.1. Patrones Creacionales

Los patrones de diseño de creación abstraen el proceso de creación de instancias. Ayudan a hacer un sistema independiente de cómo se crean, se componen y se representan sus objetos.

5.1.1. Patrón Único o *Singleton*

El **patrón de creación único** o *singleton* garantiza que sólo exista una instancia de una clase, y proporciona un punto de acceso global a ella.

Ejemplos de motivación

Aunque puedan existir varias impresoras, es importante que sólo exista una cola de impresión. La configuración de un programa debería guardarse en una única instancia a la que accedan directamente todos los componentes de programa. Una variable global puede hacer accesible un objeto, pero no evita que haya varias instancias de este objeto. El patrón *único* transfiere la responsabilidad de garantizar que sólo exista una instancia a la propia clase.

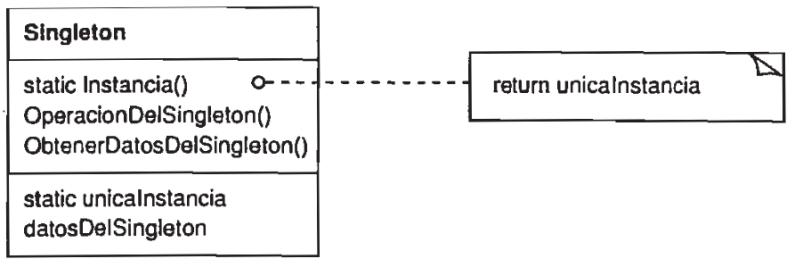


Figura 3: Estructura del patrón único [4]

5.1.2. Patrón Modelo-Vista-Controlador (MVC)

El patrón arquitectónico MVC divide una aplicación interactiva en tres componentes: **Modelo**, **Vista** y **Controlador**, separando la lógica de negocio y los datos de la interfaz de usuario y los eventos. Esta estructura facilita la reutilización de código, la separación de conceptos y el mantenimiento del sistema.

- **Modelo:** Representa y gestiona los datos y la lógica de negocio, incluyendo privilegios de acceso. Responde a solicitudes del controlador y proporciona información requerida por las vistas.
- **Vista:** Presenta los datos del modelo en un formato interactivo, generalmente como una interfaz de usuario. Solicita al modelo la información necesaria para mostrarla.
- **Controlador:** Actúa como intermediario entre la vista y el modelo. Responde a eventos del usuario, envía peticiones al modelo y actualiza la vista según sea necesario.

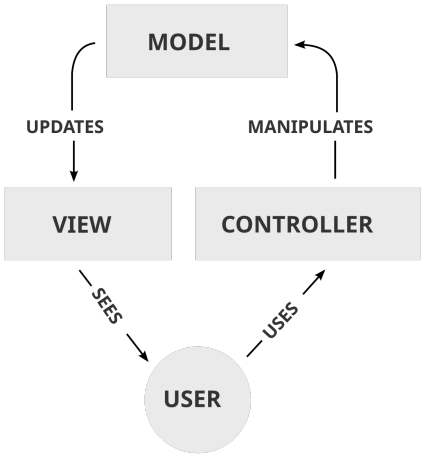


Figura 4: Colaboración entre componentes del MVC [5]

Motivación al uso de MVC

El MVC es ideal para aplicaciones con interfaces interactivas que suelen requerir cambios frecuentes (nuevas funcionalidades, soporte multiplataforma o interfaces personalizadas). Al desacoplar la lógica de negocio de la interfaz, se facilita la flexibilidad para modificar o adaptar la aplicación sin impactar el núcleo funcional, evitando la necesidad de mantener múltiples sistemas.

MVC organiza la funcionalidad en tres capas claras: procesamiento (modelo), entrada (controlador) y salida (vista), garantizando flexibilidad, escalabilidad y consistencia en el desarrollo y mantenimiento del software.

5.2. Patrones Estructurales

Los patrones de diseño estructurales se ocupan de cómo se combinan las clases y los objetos para formar estructuras más grandes.

5.2.1. Patrón Compuesto

El patrón **compuesto** (*composite*) es un patrón estructural que permite tratar de manera uniforme a objetos individuales y a composiciones de objetos. Este patrón organiza los objetos en una estructura jerárquica en forma de árbol, donde los nodos pueden ser objetos simples o compuestos, facilitando la manipulación de jerarquías complejas.

Ejemplo de motivación

En un editor gráfico, los usuarios pueden combinar elementos simples (como líneas, rectángulos y texto) para crear composiciones más grandes, como cuadros o diagramas completos. El desafío surge al tratar de manejar elementos simples y compuestos de manera uniforme, sin complicar el código. El patrón composite soluciona esto mediante una estructura recursiva donde todos los objetos, simples o compuestos, comparten una interfaz común, eliminando la necesidad de diferenciarlos explícitamente.

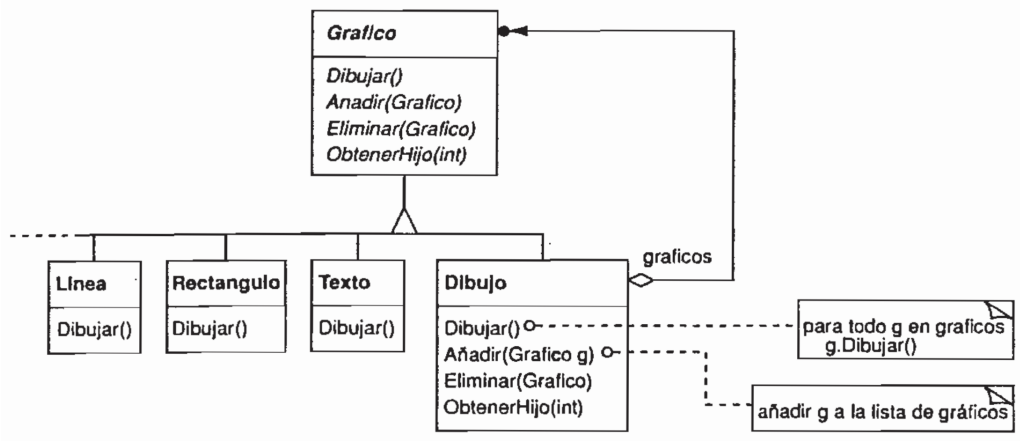


Figura 5: Ejemplo de patrón compuesto [4]

5.3. Patrones de Comportamiento

Los patrones de diseño de comportamiento tienen que ver con algoritmos y con asignación de responsabilidades a objetos. Estos patrones no describen sólo patrones de clases y objetos, sino también patrones de comunicación entre ellos.

5.3.1. Patrón Iterador

El patrón **iterador** (*iterator*) es un patrón de comportamiento que proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

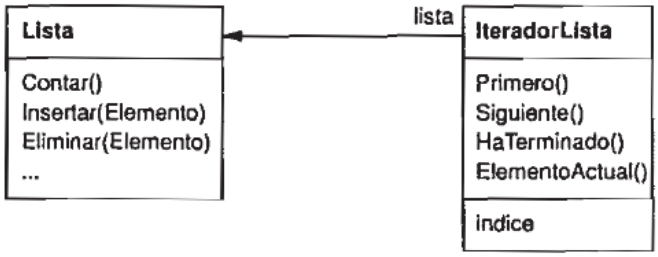
Ejemplo de motivación

Con objetos agregados como las *listas*, normalmente es preferible poder acceder a sus elementos sin exponer su estructura (implementación) interna. Además, normalmente con objetos de este tipo se desea poder recorrerlos de diferentes formas. Si quisiéramos contemplar las distintas formas de recorrer una *lista* necesitaríamos modificar la interfaz de ésta con distintas operaciones según distintos tipos de recorridos. Por otro lado, se puede dar el caso de que se quiera realizar más de un recorrido simultáneamente sobre la lista, lo que complicaría o incluso haría imposible proporcionar esta funcionalidad.

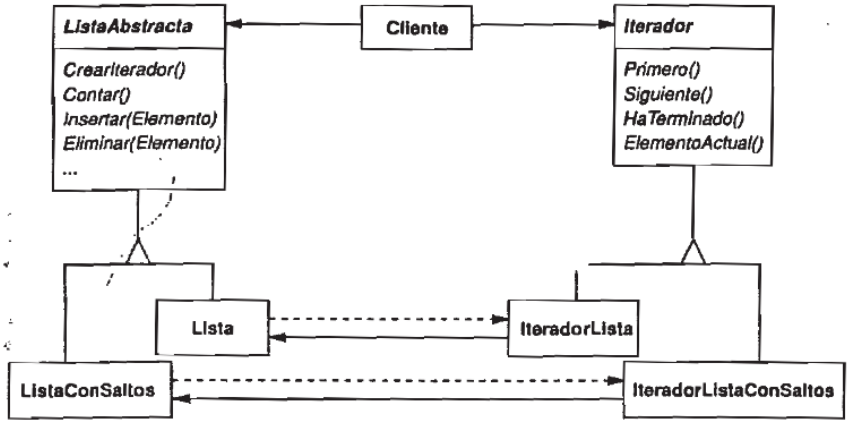
El patrón **iterador** nos permite realizar estas operaciones trasladando la responsabilidad de acceder y recorrer el objeto *lista* a un objeto **iterador**. Esta clase

definirá la interfaz de acceso a los elementos de la *lista* y será responsable de saber cuál es el elemento actual y cuáles se han recorrido ya.

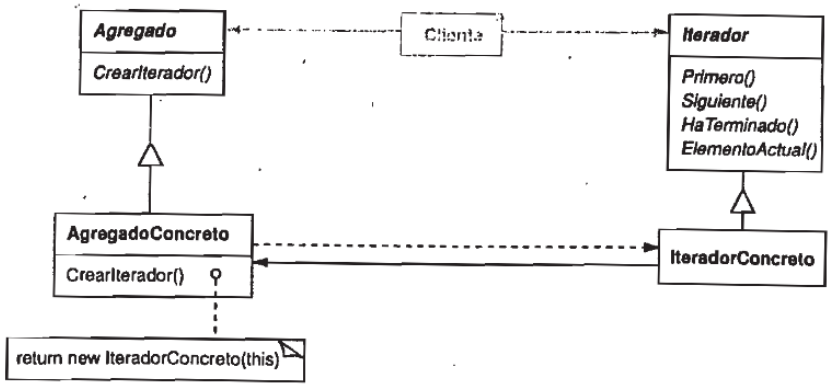
La Figura 6 muestra tres ejemplos sobre el proceso de entender el diseño del patrón iterador.



(a) Ejemplo 1



(b) Ejemplo 2



(c) Ejemplo 3

Figura 6: Diseño del patrón iterador [4]

5.3.2. Patrón Observador

El patrón observador u observer es un patrón de comportamiento que define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifiquen y actualicen automáticamente todos los objetos que depende de él. También es conocido por el nombre dependiente (Dependents) y Publicador-Subscriptor (Publish-Subscribe).

Ejemplo de motivación

En una hoja de cálculo, los gráficos y tablas que representan sus datos deben actualizarse automáticamente al cambiar la información, pero sin depender directamente de la hoja. Esto se logra mediante objetos observadores que monitorean los datos y reflejan los cambios en tiempo real, manteniendo un bajo acoplamiento entre la hoja y sus representaciones gráficas.

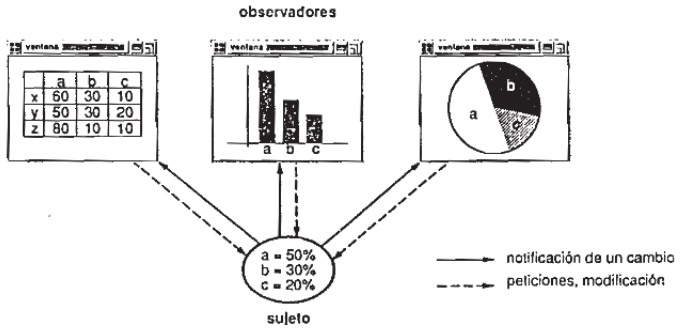


Figura 7: Ejemplo de patrón observador sobre una hoja de cálculo [4]

Referencias

[1] Roger S. Pressman, Ingeniería del Software, Un Enfoque Práctico, 9th Edition, Mc Graw Hill, 2021.

[2] Irene T. Luque Ruiz, Apuntes de la Asignatura Ingeniería del Software (2013).

[3] Los principios SOLID de programación orientada a objetos, `https://www.freecodecamp.org/espanol/news/los-principios-solid-explicados-en-espanol/`.

[4] E. Gamma, R. Helm, R. Johnson y J. Vlisides, Patrones de Diseño. Elementos de software orientado a objetos reutilizable, 2003.

[5] Modelo–vista–controlador, `https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador`.