6. Desarrollo de Software Basado en Reutilización

6.1. Introducción y objetivos

En este tema se estudian las principales características y objetivos de la reutilización del *software* junto con las técnicas utilizadas en el desarrollo de *software* basado en reutilización.

Posteriormente, este tema se centrará en los patrones para modelar la arquitectura general del sistema, como técnicas de ingeniería que facilitan la reutilización.

Por último, se estudian un conjunto de patrones específicos que permiten resolver problemas habituales en el diseño de aplicaciones. Los patrones de diseño son herramientas que permiten aplicar la reutilización de conocimiento en nuestros proyectos.

Con el estudio de este tema pretendemos alcanzar los siguientes objetivos:

* Comprender los beneficios y posibles inconvenientes de la reutilización de *software* en el desarrollo de sistemas nuevos.
* Conocer las principales técnicas de reutilización de *software.*
* Conocer los patrones arquitectónicos generales en los que se basan la mayoría de los sistemas informáticos.
* Conocer algunos patrones de arquitectura específicos, como los empleados para mejorar la tolerancia a fallos de un sistema o los empleados en sistemas distribuidos.
* Comprender la importancia de los patrones de diseño en el desarrollo de aplicaciones.
* Saber utilizar los principales patrones de diseño para la creación, estructuración y comunicación entre objetos.

6.2. La reutilización del software

El desarrollo orientado a la reutilización, aunque conceptualmente viene de lejos, solo se ha convertido en norma habitual en el entorno empresarial a partir del año 2000 (Sommerville, 2011). Progresivamente, el *software* se ha vuelto un activo más valioso para las organizaciones y se ha visto necesario adoptar estrategias que faciliten los siguientes objetivos:

* Reducir el tiempo de entrega de los sistemas.
* Aumentar la calidad del *software* desarrollado.
* Reducir los costes de desarrollo y mantenimiento de los sistemas.

Además, este movimiento se ha visto impulsado por distintos factores:

* Iniciativas *open source:* los desarrolladores disponen de enormes cantidades de código reutilizable en forma de librerías e incluso aplicaciones completas que pueden ser adaptadas a necesidades concretas.
* Estándares tecnológicos: la popularización de estándares tecnológicos facilita el desarrollo de servicios generales adaptables.

En el contexto de la ingeniería del *software,* el concepto de reutilización de *software* va más allá de la simple reutilización de líneas de código contenidas en archivos de proyectos anteriores mediante un simple copiado y pegado.

En este caso, nos estamos apoyando en conceptos que tienen que ver con (Polo Usaola, 2012):

* Diseño: técnicas de diseño orientado a objetos que permiten encapsular funcionalidades concretas en elementos potencialmente reutilizables.
* Implementación: técnicas de implementación que hacen uso de componentes existentes a distintos niveles y las posibilidades que ofrecen los lenguajes de programación modernos.
* Conocimiento: reutilización de conocimiento y experiencia acumulados durante décadas de desarrollo de proyectos *software* a nivel mundial. Aquí se pueden considerar los manuales, tutoriales y patrones de diseño como ejemplos claros de reutilización de conocimiento.

Todos estos factores, estrechamente relacionados entre sí, nos hacen ver que un aspecto importante a considerar, desde el punto de vista de ingeniería, es el enfoque adoptado en el proceso de reutilización(Polo Usaola, 2012):

* Enfoque oportunista: simplemente aprovecha recursos disponibles cuando son necesarios, aunque no se hubiera planificado previamente para ello.
* Enfoque proactivo: donde no solo se considera la utilidad del desarrollo, sino también su posible aprovechamiento futuro.

Niveles de reutilización de código

Cuando reutilizamos código existente en nuestros proyectos, según la cantidad y complejidad del código que estemos reutilizando, podemos distinguir tres grandes niveles de granularidad, según Sommerville (2011, p. 426):

* Grano grueso: reutilización a nivel de aplicación. Hay disponibles aplicaciones completas que pueden ser adaptadas, con mayor o menor esfuerzo de modificación, a necesidades específicas. Muchos *frameworks* de desarrollo también ofrecen gran cantidad de componentes y técnicas específicas de desarrollo, y se pueden incluir en esta categoría.
* Grano medio: reutilización a nivel de componentes. Aquí se puede encontrar gran variedad de tamaños, desde subsistemas completos (como un subsistema de autenticación) hasta objetos individuales complejos.
* Grano fino: reutilización a nivel de objetos y funciones. Es la más antigua y generalizada, y es la que utilizamos cuando incluimos en nuestro código una librería con utilidades o definiciones de tipos y funciones listas para ser utilizadas.

En los proyectos generalmente se utilizan varios de estos niveles de reutilización y los niveles superiores engloban siempre a los inferiores.

### **6.3. Técnicas de reutilización**

Los **enfoques más antiguos** permitieron la reutilización de rutinas encapsuladas en simples funciones, que eran recopiladas en librerías listas para su utilización. Más adelante, el paradigma de la orientación a objetos permitió reutilizar elementos de complejidad y granularidad creciente, como las clases. A medida que se acumuló conocimiento y experiencia, la comunidad de ingeniería de software fue desarrollando patrones de arquitectura y de diseño que evitaban tener que enfrentarse desde cero a problemas de diseño habituales. Más**recientemente,** tenemos a nuestra disposición componentes y subsistemas enteros, frameworks de desarrollo y servicios que podemos aprovechar en nuestros proyectos.

### **Los patrones arquitectónicos**

El diseño de la arquitectura de un sistema es una representación de su estructura general al más alto nivel.

En este nivel se omiten detalles específicos y se trabaja con un **nivel de abstracción** lo suficientemente elevado como para que no sean relevantes los detalles de implementación.

Los buenos diseños arquitectónicos favorecen los siguientes aspectos (Polo Usaola, 2012):

* Permiten identificar las relaciones necesarias entre distintos sistemas o subsistemas.
* Como consecuencia de lo anterior, se favorece la reutilización de elementos existentes y permiten estudiar la manera de mejorar o adaptar otros que no pueden ser utilizados en su estado actual.
* Es una potente herramienta de documentación del código que ayuda a comprender su funcionamiento general en un solo golpe de vista.
* A lo largo de los años se han producido multitud de patrones arquitectónicos que dan respuesta a problemas concretos.

### **Los patrones de diseño**

Podemos considerar un patrón de diseño como una solución general para el diseño de un software que debe resolver un tipo de problema frecuente.

Es la frecuencia de la necesidad de esta implementación lo que ha hecho que el conocimiento acumulado haya ido definiendo ciertas maneras de **abordar el problema** que han demostrado éxito en la práctica.

**Patrones**

Para que una determinada solución pueda ser **considerada un patrón** se deben dar las siguientes circunstancias:

* Que haya sido aplicada y probada con éxito un número suficiente de veces.
* Que pueda ser descrita con la suficiente generalidad como para permitir su aplicación o reutilización en contextos diferentes.

Además, la mayoría de los patrones de diseñoconsideran intrínsecamente la reutilización como un**valor deseable,** de manera que, cuando aplicamos un patrón, no solamente estamos reutilizando un diseño, sino que a su vez el patrón definirá ciertos elementos (clases) reutilizables en nuestra aplicación. Esto, en ocasiones, puede precisar que el patrón requiera la implementación de clases o elementos adicionales, intermedios, que a veces pueden repercutir negativamente en la complejidad del sistema y su mantenimiento.

### **Las librerías**

Una librería de código es un conjunto de funciones reutilizables contenidas en uno o varios ficheros.

Además, estas funciones suelen estar agrupadas teniendo en cuenta que aportan **funcionalidades relacionadas** (funciones matemáticas, funciones de procesamiento de imagen, funciones para creación de gráficos, etc.).

Cuando utilizamos este tipo de elementos en nuestros proyectos, generalmente, el código de la librería se encuentra alojado de manera **externa** al repositorio de nuestro proyecto, en directorios diferentes.

Aparecen dos enfoques fundamentales a la hora de reutilizar este código:

* **Enlace estático:** el código que estamos utilizando de las librerías se combina con el código de nuestro programa, generando un único ejecutable de gran tamaño. Esto hace que podamos tener código procedente de librerías duplicado en nuestro sistema, dentro de los correspondientes ejecutables.
* **Enlace dinámico:** las librerías de enlace dinámico son aquellas que se encuentran instaladas en el sistema operativo de manera separada del software que las utiliza. Cuando un ejecutable necesita utilizar alguna función de estas librerías, la solicita al sistema operativo. En este caso es posible que programas diferentes reutilicen código de las mismas librerías, y este código no está duplicado.

Sin embargo, la utilización de librerías de enlace dinámico puede provocar inconvenientes:

* **Problemas de compatibilidad con distintas versiones de una librería.** Esto ocurre cuando en un sistema tenemos varios programas que utilizan una misma librería, pero no todos pueden con la misma versión.
* **Problemas de mantenimiento.** Lo anterior obliga a que generalmente en un mismo sistema coexistan varias versiones de la misma librería. Además, la desinstalación manual de alguna de ellas puede provocar que algunas aplicaciones dejen de funcionar.

Aunque nos hemos referido a las librerías como conjuntos de funciones reutilizables, es cierto que podemos aplicar el mismo término (librería o biblioteca) a cualquier conjunto de recursos almacenados en archivos que podemos reutilizar en nuestros programas, incluyendo imágenes, sonidos, objetos, etc.

### **Las clases**

Como se ha visto, la aparición del **paradigma de la programación** orientada a objetos (POO) y las características —como la abstracción, la herencia, el polimorfismo y la encapsulación— están directamente relacionadas con la búsqueda de la reutilización de código.

### **Los servicios**

Según Polo Usaola (2012), un **servicio** es «una funcionalidad que un proveedor de servicios ofrece a sus clientes de forma remota»(p. 65) y puede ser considerado un componente más del sistema, que se utiliza a través de las operaciones que ofrece su interfaz.

El W3C (2004) los define como:

«[…] un sistema software destinado a soportar la interacción entre máquinas a través de una red, que ofrece una interfaz descrita en un lenguaje procesable por máquinas (específicamente WSDL), y con el que otros sistemas interactúan utilizando mensajes SOAP, generalmente transportados sobre HTTP y con serialización XML en combinación con otros estándares web».

Según esta definición:

* **XML**(eXtensible Markup Language) es el formato utilizado para contener los datos que se intercambian.
* **SOAP**(Simple Object Access Protocol) es el protocolo utilizado para transferir los datos.
* **WSLD**(Web Services Description Language) describe los servicios disponibles.

Sin embargo, hoy en día la variedad de servicios web, protocolos utilizados y formatos de intercambio es tan amplia que podemos considerar como **servicio web** cualquier funcionalidad que nos ofrece un servidor remoto. Cada vez es más habitual encontrar API que proporcionan sus servicios en formatos como JSON, más ligero que XML, adoptando el estilo arquitectónico REST (RESTful APIs).

### **Los componentes**

Podemos definir un **componente,** según Sommerville (2011) como «una unidad de software independiente que puede organizarse con otros componentes para crear un sistema de software» (p. 455).

Heineman y Councill (2001), dan una definición más precisa y lo consideran un **elemento de** software que se conforma a un modelo de componentes estándar y puede desplegarse y componerse independientemente sin modificación, de acuerdo con un estándar de composición.

Existe un consenso general de que, en todo caso, un componente es un elemento que se incluye en el sistema ofreciendo un conjunto de interfaces estandarizadas, lo que lo diferencia de un servicio externo.

Estas interfaces suponen un **contrato** entre ese elemento y el resto del sistema, que lo ve como una «caja negra» (y, de hecho, es habitual no tener acceso al código fuente del componente utilizado).

Podemos resumir del siguiente modo las características de los componentes (Sommerville, 2011):

* **Estandarizado:** los componentes se ajustan a modelos estandarizados que definen sus interfaces, metadatos y documentación, y facilitan su integración.
* **Independiente:** un componente es un elemento aislado, susceptible de ser intercambiado por otro componente diferente, si es necesario, gracias a la estandarización. En caso de que el componente necesite a su vez de un servicio externo debe especificarse a través de una interfaz de requisitos o dependencias.
* **Componible:** las interfaces públicas que ofrece deben facilitar su integración con otros servicios, permitiendo el acceso a información y operaciones.
* **Implementable:** debe existir una plataforma tecnológica o framework general de desarrollo que permita su utilización. En el caso de que el componente sea un servicio externo, será responsabilidad del proveedor ofrecer esta infraestructura.
* **Documentado:** deben ofrecer documentación adecuada para que los potenciales usuarios puedan valorar su utilidad y si cumplen con sus necesidades.

Cualquier componente tiene en general dos **interfaces relacionadas** (Figura 1):

* Interfaz **«proporciona»**, que define los servicios ofrecidos a través de una API.
* Interfaz **«requiere»**, que indica los servicios de otros componentes de los cuales depende y, si no están disponibles, el componente no funcionará.

A diagram of a computer component

Description automatically generated

Figura 1. Interfaces de un componente. La interfaz «proporciona» es siempre necesaria. Fuente: elaboración propia.

### **Los frameworks de desarrollo**

A pesar de los beneficios de la programación orientada a objetos, la reutilización de objetos de manera**individual** resulta difícil. Esto, según Sommerville (2011), se debe a que a menudo su tamaño es demasiado pequeño y su propósito tan específico, que puede resultar más simple su implementación desde cero que su adaptación (p. 431).

Sin embargo, los frameworks de desarrollo ofrecen un entorno idóneo que simplifica y guía la reutilización de componentes de grano más fino.

Para Schmidth et al. (2004), un framework de desarrollo es «un conjunto integrado de artefactos de software (como clases y componentes) que colaboran proporcionando una arquitectura reutilizable para un conjunto de aplicaciones relacionadas» (p. 68).

Permiten desacoplar las partes de código que dependen del dominio de aplicación específico, de aquellas que no lo son y son comunes a otras aplicaciones de la misma categoría. Permiten mejorar la **extensibilidad y portabilidad** del sistema del siguiente modo (Schmidth et al., 2004):

* **Reutilización del diseño:** la mayoría de frameworks adoptan determinados patrones útiles en su contexto de desarrollo específico y, de este modo, ofrecen al desarrollador una guía de cómo debe diseñarse la aplicación.
* **Reutilización de implementaciones:** los frameworks también van acompañados de librerías de clases y funciones que resuelven problemas habituales de desarrollo (por ejemplo, la interacción con bases de datos o el diseño de la interfaz a partir de plantillas).
* **Reutilización de validaciones:** Los elementos que reutilizamos al aplicar el framework ya han sido validados. Además, en muchos casos los frameworks integran procedimientos de validación específicos que ayudan al desarrollador.

Sin embargo, la utilización de un framework puede resultar compleja en ocasiones. Identificamos algunos **posibles problemas** (Johnson, 1997; Sommerville, 2011):

* Dificultad de aprendizaje. Aprender a manejar algunos frameworks con agilidad puede llevar meses.
* Su desarrollo es complejo y requiere programadores de gran experiencia.
* Existe gran variedad de este tipo de frameworks, lo cual dificulta su elección.
* La depuración de las aplicaciones puede requerir conocimientos avanzados sobre el funcionamiento interno del framework, lo cual puede resultar complejo.

Una clase particular de framework, muy popular hoy en día, es la que permite desarrollar **aplicaciones web.** En general, se basan en el **patrón Modelo-Vista-Controlador** (Gamma, Helm, Johnson y Vlissides, 2003). A su vez, suelen incluir otros frameworks más específicos para resolver problemas concretos.

La mayoría de los frameworks web ofrecen las siguientes características (Sommerville, 2011):

* **Seguridad,** ofreciendo clases para resolver problemas habituales como la autenticación de usuarios y el control de acceso.
* **Páginas web dinámicas,** soportando algún sistema de plantillas que permita dotarlas de contenidos específicos, generando así el HTML final.
* **Soporte para bases de datos.** Aunque, en general, los frameworks no están sujetos al empleo de una base de datos en particular, ofrecen clases que facilitan la interacción con diferentes opciones.
* **Interacción con los usuarios,** simplificando la utilización de otros frameworks de desarrollo front-endque permiten crear páginas más interactivas.

### **Reutilización de aplicaciones**

En ocasiones, es posible reutilizar un sistema software íntegro en su estado actual o con mínimas modificaciones para dar respuesta a nuestras necesidades.

Los productos COTS (Commercial-Off-The-Shelf) se pueden adaptar a las necesidades del cliente sin cambiar el código fuente del sistema (Sommerville, 2011, p. 440).

Este enfoque se ha vuelto cada vez más habitual y prácticamente todas las herramientas ofimáticas, de correo electrónico, sistemas ERP y CMS, y productos en el lado del servidor pertenecen a esta categoría.

Este enfoque ofrece algunas ventajas y problemas.

* **Ventajas:**
  + La implementación del sistema es mucho más rápida que con otras alternativas.
  + Se puede probar la funcionalidad del proyecto y su ajuste a nuestros requisitos.
  + Se utiliza un software existente y probado.
  + El mantenimiento del producto y sus actualizaciones dependen del proveedor.
* **Desventajas:**
  + Su empleo puede obligar a modificar los procesos internos del consumidor.
  + Su elección puede ser difícil por la variedad de oferta o la falta de documentación.
  + Existen intereses comerciales que pueden afectar a la mejor elección.
  + El control de soporte y las decisiones sobre la evolución del sistema están en el proveedor, que puede decidir abandonar el desarrollo o introducir modificaciones que ocasionen dificultades para el cliente.

Encontramos dos categorías dentro de este enfoque de reutilización, **sistemas de solución** y**sistemas integrados**, cuyas características se resumen en la Tabla 1.

A table with text on it

Description automatically generated

Tabla 1. Comparativa entre sistemas COTS: sistemas de solución y sistemas integrados. Fuente: elaboración propia.

Ejemplo de sistema COTS

Supongamos que una empresa tiene la necesidad de crear un blog empresarial. Podría adoptar dos enfoques alternativos dentro de una solución COTS: contratar un blog en WordpPress.com y configurarlo según sus necesidades, seleccionando la plantilla y el conjunto de plugins más adecuado; o descargar el CMS desde WordPress.org e instalarlo en su propia infraestructura, configurando la base de datos y otros elementos y subsistemas que permitan dar solución a sus necesidades.

En la Figura 2 podemos observar las diferencias que existen entre:

* La construcción de un sistema completamente a medida, posiblemente empleando algún elemento de granularidad fina existente (como librerías o paquetes de clases, o incluso un framework de desarrollo).
* La creación del sistema a partir de componentes bajo un modelo CBSE (que pueden ser a su vez productos COTS).
* La utilización de un producto COTS, que solo requiere su adaptación a las necesidades del cliente, con un mínimo esfuerzo de desarrollo.

Diagram of components of a computer component

Description automatically generated

Figura 2. Diferencias entre el desarrollo de un sistema a medida, su construcción a partir de componentes y la utilización de un sistema COTS. Fuente: Esri, 2011.

### **6.4. Patrones de arquitectura**

Para Sommerville (2011), los patrones arquitectónicos son una manera de:

«[…] representar, compartir y reutilizar el conocimiento sobre los sistemas de software [que se manifiesta como una] descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos» (p. 156).

Es decir, un patrón ha debido mostrar su **utilidad** en desarrollos previos e, idealmente, debería ir acompañado de **recomendaciones** sobre su aplicabilidad, fortalezas y debilidades.

### **Arquitecturas cliente-servidor**

En estos sistemas, la lógica se distribuye entre dos tipos de máquinas, clientes y servidores, conectados por algún tipo de middleware (Polo, 2012).

Ambos tipos de sistemas pueden tener **arquitecturas internas** muy diferentes. Estas arquitecturas son habituales en los sistemas distribuidos, pero también pueden aplicarse a procesos que se ejecutan en la misma máquina.

Los elementos fundamentales son:

* **Servidores:** un conjunto de servidores que proporcionan servicio a otros componentes (servidores de impresión, de archivos, de datos, etc.).
* **Clientes:** un conjunto de clientes que solicitan servicios a los servidores. Generalmente, existen varios clientes distribuidos trabajando simultáneamente con el mismo servidor, pero sin conocerse entre sí.
* **Red:** una red o middleware que permite la conexión entre clientes y servidores.

En la Figura 3 se muestra el ejemplo de un museo virtual como **sistema multiusuario,** donde además se aprecia que los distintos servidores a los que acceden los clientes se comportan a su vez como clientes frente a sus respectivos repositorios de datos, definiéndose un modelo de dos capas en cada uno de los servidores.

A diagram of a diagram

Description automatically generated

Figura 3. Ejemplo de patrón cliente–servidor en un museo virtual. Fuente: elaboración propia.

La Tabla 2 resume las **principales características de este patrón arquitectónico.** Esta es una manera habitual, aunque resumida, de describir los patrones; al menos se espera una descripción, algún tipo de ejemplo práctico, recomendaciones sobre su aplicación y alguna valoración crítica, indicando las principales ventajas e inconvenientes de su empleo.

A blue and white text box

Description automatically generated

Tabla 2. Descripción del patrón cliente–servidor. Fuente: elaboración propia.

En función de cómo esté distribuida la lógica entre los diferentes nodos, podemos encontrar diferentes **situaciones particulares,** como se aprecia en la Tabla 3. Interesa resaltar la diferencia entre:

* **«Clientes ligeros»,** que únicamente contienen lógica de presentación.
* **«Clientes pesados»,** que, además, realizan operaciones adicionales con los datos y, en muchos casos, almacenan también información.

A diagram of a customer service

Description automatically generated

Tabla 3. Posibles repartos de responsabilidades en arquitecturas cliente–servidor. Fuente: adaptada a partir de Polo Usaola, 2012.

### **Arquitecturas de tuberías y filtros**

Las arquitecturas de «tuberías y filtros» (pipes and filters) modelan sistemas cuyos componentes principales son nodos especializados en aceptar datos de entrada, procesarlos y producir unos resultados con ellos.

Estos nodos, o filtros, se interconectan entre sí a través de algún mecanismo de comunicación (las tuberías), de manera que la representación global del sistema adopta la forma de un **grafo dirigido.**

Una característica importante de estos sistemas es que los nodos son independientes entre sí, no comparten información de estado ni tienen por qué conocer al resto de nodos de la red.

Un ejemplo clásico de este tipo de sistema son los **compiladores de lenguajes** y, en general, cualquier sistema de procesamiento de lenguaje o de señal, como se muestra en la Figura 9. Ejemplo de patrón tuberías y filtros en un esquema simplificado de compilación y enlace.

A diagram of a diagram

Description automatically generated

Figura 9. Ejemplo de patrón «tuberías y filtros» en un esquema simplificado de compilación y enlace. Fuente: elaboración propia.

En este tipo de sistemas es fácil la **reutilización** de componentes, o filtros, siempre y cuando los protocolos de comunicación entre ellos sean respetados. Además, dada la independencia de los componentes el **mantenimiento** suele ser sencillo.

Las características principales de este patrón se resumen en la Tabla 4.

A blue and white text on a blue background

Description automatically generated

Tabla 4. Descripción del patrón «tuberías y filtros». Fuente: elaboración propia.

### **Arquitecturas multicapa**

En este caso, el sistema es susceptible de ser descompuesto en **diferentes capas** (‘tiers’o ‘layers’, en inglés), de manera que cada una de ellas se especializa en un conjunto de **responsabilidades** y ofrece **servicios** a las capas situadas en el nivel inmediatamente superior.

Idealmente, cada capa solo conoce a las adyacentes y solo ofrece servicios a la que está situada inmediatamente por encima, pero pueden encontrarse variaciones de este modelo.

Este tipo de arquitectura soporta el **enfoque de modelo incremental de sistemas,** comenzando por el desarrollo de las capas inferiores, de manera que sus servicios estén disponibles cuando comienzan a desarrollarse las superiores. También es apropiado cuando se necesita establecer distintos niveles de control de acceso a un sistema. Los sistemas operativos siguen también este patrón arquitectónico.

En la arquitectura de capas de la plataforma Android, Figura 10, encontramos un Kernel Linux en el nivel inferior y, sobre él, se van construyendo servicios hasta llegar a la capa de API, que utilizan los programadores de aplicaciones.

A screenshot of a web page

Description automatically generated

Figura 10. Tres ejemplos de arquitectura de capas. La arquitectura de comunicación del modelo OSI, un modelo genérico para aplicaciones web, y la arquitectura de capas de la plataforma Android. Fuente: Adaptada a partir de Android, 2020.

Las características principales de este patrón se resumen en la Tabla 5.

A screenshot of a computer

Description automatically generated

Tabla 5. Descripción del patrón de arquitectura en capas. Fuente: elaboración propia.

### **Arquitectura de repositorio**

Este tipo de arquitectura es adecuada para sistemas que utilizan grandes cantidades de información que comparten diferentes subsistemas. Generalmente existen componentes generadores de datos y otros que los consumen, de manera que los datos quedan almacenados en un repositorio común.

En la Figura 11 se muestra un ejemplo de este patrón para el caso de un sistema de gestión hospitalaria. Todos los subsistemas comparten información sobre los pacientes, que está almacenada y centralizada en un único servidor. En este servidor se puede definir una **arquitectura interna orientada a la seguridad**como una arquitectura de capas. Este modelo no presenta diferencias respecto del cliente-servidor, pero está orientado a la concentración de los datos en el repositorio común.

A diagram of a diagram

Description automatically generated

Figura 11. Ejemplo de una arquitectura de repositorio para un sistema de gestión hospitalaria. Fuente: elaboración propia.

Esta es una manera eficiente decompartir**grandes cantidades de datos,** pero todos los participantes deben ponerse de acuerdo en un **modelo de datos común.**

El repositorio se encuentra lógicamente centralizado, aunque a nivel físico pueda estar distribuido en diferentes máquinas. Esta posible distribución, orientada en general a facilitar la escalabilidad del sistema, es uno de los puntos que plantea problemas en este modelo.

Lógicamente, al tener los datos centralizados en un único lugar, este es precisamente el punto más vulnerable del sistema.

Las características principales de este patrón se resumen en la Tabla 6.

A blue and white text box

Description automatically generated

Tabla 6. Descripción del patrón de arquitectura de repositorio. Fuente: elaboración propia.

### **Arquitecturas tolerantes a fallos**

Según Sommerville (2011), la confiabilidad de un sistema es «una propiedad agregada que toma en cuenta la protección, fiabilidad, disponibilidad, seguridad y otros atributos del sistema, y refleja el grado en el que los usuarios pueden confiar en él» (p. 735).

Los **sistemas críticos** —como los de control de proceso, sistemas médicos o sistemas de control de vuelo— requieren de técnicas específicas para mejorar su confiablidad, y, entre ellas, están las que refuerzan la arquitectura del sistema. Esto pasa por analizar cuidadosamente la**tolerancia a fallos** incluyendo, en general, componentes y mecanismos redundantes. Vamos a ver aquí un par de ejemplos de estas arquitecturas.

**Sistemas de protección**

Un sistema de protección está especializado en **monitorizar el correcto funcionamiento** de otro sistema al que está asociado. Son habituales en los sistemas de control de proceso (como la industria química) o control de equipo (como los procesos de fabricación o el control de vehículos).

El sistema de protección funciona en paralelo con el controlador de proceso, monitorizando tanto las **condiciones ambientales** como el **estado del propio proceso.** En caso de detectar alguna desviación sobre el comportamiento esperado, puede tomar el control del sistema o, al menos, provocar algún tipo de aviso al exterior. En la Figura 12 podemos observar un ejemplo de ello.

A diagram of a system

Description automatically generated

Figura 12. Ejemplo de un sistema de protección. Fuente: elaboración propia.

El sistema de protección adquiere información del entorno a través de un**conjunto independiente de sensores.** También monitoriza otras variables del sistema, como el estado de los sensores y actuadores de proceso, y el propio sistema de control. Puede contar con un sistema de actuadores independiente al del proceso (por ejemplo, un sistema de detención de emergencia en un vehículo no tripulado). En caso de ser necesario, puede entrar en acción y evitar un fallo catastrófico del sistema.

**Programación de varias versiones**

En este caso, se desarrollan **varias versiones del mismo sistema** que se ejecutan en**máquinas diferentes.**Partiendo de una especificación común, las distintas versiones son desarrolladas por diferentes equipos, así se llega a implementaciones alternativas.

Las salidas que generan los diferentes sistemas se comparan en un **sistema de votación,** rechazando aquellas salidas que son inconsistentes o que no se producen a tiempo. Este tipo de protección es habitual en los sistemas de señalización ferroviaria, sistemas de control de aeronaves y protección de reactores (Sommerville, 2011, p. 353).

A diagram of software

Description automatically generated

Figura 13. Ejemplo de protección con tres versiones de software.Fuente: elaboración propia.

### **Arquitecturas de sistemas distribuidos**

Actualmente, la mayoría de los grandes sistemas son **sistemas distribuidos,** donde el cómputo se reparte entre un gran número de máquinas y se contrastan con los sistemas centralizados cuyos componentes se ejecutan en un solo computador.

Para Tanenbaum y Steen (2007), un sistema distribuido es una colección de computadores independientes que aparecen ante el usuario como un solo sistema coherente.

Según Sommerville (2011), estos sistemas ofrecen **múltiples ventajas** (p. 480):

* **Compartición de recursos:** puesto que los diferentes subsistemas ofrecen recursos a otros a través de una conexión de red (almacenamiento, archivos, impresoras, etc.).
* **Apertura empleando protocolos estandarizados:** que facilita las combinaciones de hardware y software de proveedores diferentes.
* **Concurrencia:** al poder ejecutarse diferentes procesos de manera simultánea en diferentes máquinas, coordinadas o no entre sí.
* **Escalabilidad:** al poderse ampliar el sistema agregando más recursos en función de la demanda existente, siempre que la infraestructura de red lo permita.
* **Tolerancia a fallos:** puesto que la distribución de recursos facilita la redundancia y, en caso de fallar un nodo, en la mayoría de los casos se produce una degradación de la calidad del servicio. Los fallos completos del servicio se deben a fallos de red.

Estos sistemas son más complejos que los centralizados, tanto en su diseño como en su implementación. En general, según Sommerville (2011), se debe encontrar un **equilibrio** entre diferentes **requisitos no funcionales** —como el rendimiento, la confiabilidad, la seguridad y la manejabilidad del sistema (p. 490)—. Con el tiempo, han surgido algunos modelos arquitectónicos que se ocupan de manera específica de este tipo de problemas.

**Arquitecturas maestro-esclavo**

Existen muchos sistemas con fuertes restricciones de tiempo real, en los que el cómputo se distribuye a diferentes escalas entre distintos nodos de procesamiento.

Puede haber nodos especializados en la adquisición de información del entorno, nodos especializados en controlar a los actuadores, nodos especializados en cómputo, etc.

En este tipo de sistemas es habitual la presencia de un **proceso «maestro»,** responsable de gestionar la comunicación y coordinar al resto de participantes. Se trata de una arquitectura centralizada en la que el maestro toma la forma de un nodo cliente-servidor; actúa como cliente frente a otros procesos, como el control de sensores (pidiendo información) o el control de actuadores (enviando consignas de operación), y sirve información a otros procesos como los clientes de las consolas de operador.

En la Figura 14 se muestra un ejemplo de esta arquitectura, correspondiente a un sistema de gestión de tráfico.

A diagram of a process

Description automatically generated

Figura 14. Ejemplo de arquitectura maestro–esclavo: sistema de control de tráfico. Fuente: Sommerville, 2011.

**Arquitecturas cliente-servidor de varios niveles**

El modelo más simple de arquitectura cliente-servidor fue presentado al comienzo del tema.

El patrón cliente-servidor predomina en sistemas distribuidos y, en el caso más básico, presenta únicamente dos niveles lógicos: existe un único servidor y un conjunto más grande de clientes que consumen sus servicios.

Este modelo es compatible con una arquitectura multicapa, pero presenta el problemade que **todas las capas** (presentación, procesamiento de aplicación, gestión de datos y base de datos) **se mapean en dos únicos sistemas cómputo.** Lo que perjudica el rendimiento y la escalabilidad del sistema, especialmente cuando elegimos un modelo de cliente ligero.

Las arquitecturas**cliente-servidor de varios niveles** son aquellas que distribuyen los diferentes servidores, con capacidades específicas, en diferentes niveles físicos.En la Tabla 7 se resumen las principales aplicaciones de cada modelo.

A blue and white screen with text

Description automatically generated

Tabla 7. Variantes del patrón cliente–servidor en sistemas distribuidos. Fuente: adaptada a partir de Sommerville, 2011.

Ejemplos de aplicaciones apropiadas para emplear un **cliente ligero** son los procesos de compilación en lote (intensivos en cómputo) o la navegación web (intensivo en datos). En el caso de los **clientes pesados,**encontraríamos aplicaciones como la visualización de datos (intensiva en cómputo en el lado del cliente).

**Arquitecturas de componentes distribuidos**

La arquitectura multicapa, entendida en este contexto como una arquitectura cliente–servidor multinivel, permite que cada una de ellas se implemente como un **servidor separado.** Esto puede **plantear problemas de diseño;** no siempre está claro qué servicios incluir en cada capa, y puede plantear **problemas de escalabilidad** a la hora de replicar los servidores de cada capa a medida que se agregan clientes.

Un enfoque más flexible consiste en diseñar el sistema como un conjunto de servicios comunicados a través de un middleware, como se muestra en la Figura 15. Aquí se ha adaptado el ejemplo anterior de la Figura 3 que se enfocaba en el patrón cliente-servidor de dos niveles, con cada servidor modelado en una arquitectura de dos capas. La nueva arquitectura de componentes distribuidos resulta más flexible, y parece más sencillo poder agregar nuevas bases de datos al sistema.

A diagram of a diagram

Description automatically generated

Figura 15. Ejemplo de arquitectura de componentes distribuidos para un museo virtual. Fuente: elaboración propia.

En este patrón identificamos una serie de beneficios y desventajas:

* **Ventajas:**
  + **Simplificación del diseño,** pues no obliga a tomar la decisión de dónde y cómo deben proporcionarse los servicios, o en qué capa. Los servicios simplemente se conectan a la red.
  + **Aumento de la flexibilidad.** Es sencillo agregar nuevas funcionalidades o servicios al sistema a medida que resultan necesarios.
* **Desventajas:**
  + La comprensión del sistema puede resultar más complicada frente a una arquitectura multicapa que resulta más atractiva.
  + Un middleware estandarizado compatible con la filosofía de componentes no ha sido aceptado por la comunidad. Algunos fabricantes como Microsoft o Sun desarrollan sus propias soluciones incompatibles entre sí, pero el empleo de un middleware aumenta la complejidad de diseño de los componentes.

Por estos motivos las arquitecturas de componentes distribuidos son a menudo **sustituidas** por arquitecturas orientadas a servicios.

CORBA (Common Object Request Broker Architecture)

La arquitectura común de intermediario de peticiones de objetos fue propuesta por el Object Management Group en la década de 1990 como middleware que permitía la comunicación entre componentes distribuidos (Sommerville, 2008).

Aparecieron varias implementaciones, pero nunca llegó a ser plenamente aceptado. Pronto fue sustituido por tecnologías propietarias (como Enterprise Java Beans o .NET) o reemplazado por arquitecturas orientadas a servicios.

**Arquitecturas P2P**

Las **arquitecturas P2P** (peer-to-peer), o entre pares, contienen varios sistemas que se intercomunican entre sí en condiciones de igualdad. En este caso, todos los elementos del modelo son**homogéneos,** tienen características y funcionalidades similares. Esto no quiere decir que sean idénticos en su implementación interna, pero sí que, al menos desde el punto de vista externo,ofrecen las mismas interfaces, tienen las mismas necesidades y ofrecen los mismos servicios.

Dicho con otras palabras, todos los nodos son simultáneamente clientes y servidores para el resto, y las interfaces son idénticas. Esta interfaz común es susceptible de ser implementada a través de algún componente reutilizable, que simplemente se integra en la implementación de los distintos componentes concretos del sistema P2P.

La ventaja de este enfoque es que permite aprovechar mejor los **recursos computacionales disponibles en la red,** al no hacer una distinción clara entre nodos servidores y nodos cliente. Esta tecnología se ha venido utilizando más frecuentemente en el ámbito personal que en el empresarial, en forma de sistemas para compartir archivos basados en protocolos como Gnutella o BitTorrent, o sistemas de mensajería como Skype.

A nivel empresarial e institucional también se utilizan en aplicaciones que requieren un **cómputo intensivo**como alternativa a la compra de un **único sistema** hardware de alto rendimiento. Se distribuye así el procesamiento entre **infinidad de nodos equivalentes,** que aprovechan las capacidades locales de las máquinas de los usuarios de la red. Ejemplo de ello es el proyecto SETI@home (Universidad de California, 2018), que pretende encontrar señales de inteligencia extraterrestre (Search for Extra Terrestrial Intelligence) analizando de manera distribuida datos adquiridos por radio telescopios.

Sin embargo, estos sistemas deben ofrecer algún mecanismo que permita a los nodos descubrirse y conectarse entre sí. En un esquema P2P puramente descentralizado, todos los nodos deberían comportarse como conmutadores de comunicaciones, actuando de intermediarios en los procesos de conexión.

### **6.5. Patrones de diseño**

El apartado Patrones de arquitectura se centró en el estudio de los patrones de arquitectura que permiten modelar al más alto nivel los componentes principales de un sistema, así como las relaciones que se establecen entre ellos.

Aquí nos centraremos en soluciones más específicas para problemas de desarrollo habituales, mucho más cercanas al dominio de implementación del sistema.

Los **patrones de diseño** fueron definidos como:

«[…] la forma de describir un problema que se da una y otra vez en nuestro entorno, y que especifica el núcleo de la solución para ese problema, de tal forma que se pueda utilizar esa solución un millón de veces más, sin que se tenga que hacer de la misma manera más de una vez» (Alexander et al., 1977).

Es decir, cada patrón expresa una relación entre un contexto específico, un problema y una solución.

Para poder considerar una **solución como un patrón de diseño** deben darse al menos dos características (Polo Usaola, 2012):

* **Debe ser repetible,** habiendo mostrado su efectividad en más de una ocasión. Por tanto, la solución no solo debe resultar válida para una tecnología o problema de un dominio de aplicación concreto.
* **Su descripción debe ser lo suficientemente genérica** como para que pueda ser aplicada en contextos tecnológicos diversos.

En algunos casos, los patrones de diseño ofrecen soluciones que, en apariencia, **complican excesivamente**el diseño del sistema (por ejemplo, introduciendo elementos adicionales que a priori no parecen indispensables). Por este motivo, el abuso de su empleo (o su utilización incorrecta) puede hacer que el sistema se vuelva más difícil de comprender y mantener.

Sin embargo, en términos generales, los patrones de diseño persiguen aumentar la cohesión y reducir el acoplamiento de los sistemas, favoreciendo prácticas deseables como la reutilización de software.

Tal es la importancia que cobran los patrones de diseño como técnicas de reutilización de software y resolución rápida de problemas habituales, que este tipo de conocimiento es recopilado y publicado en diferentes manuales y catálogos.

Estos catálogos utilizan una **estructura estandarizada** y generalmente definen, entre otros, los siguientes elementos para cada uno de los patrones:

* Nombre del patrón, generalmente en inglés.
* Clasificación del patrón (de creación, de estructura o de comportamiento).
* Intención o descripción del problema que intenta resolver.
* Motivación o explicación de escenarios prácticos que justifican la posible aplicación del patrón.
* Estructura, empleando diagramas de clases que representan los objetos que intervienen en el patrón y sus relaciones.
* Consecuencias positivas y negativas de su empleo. Describe los resultados que se obtienen tras aplicar el patrón.
* Implementación, ofreciendo indicaciones para ello.
* Código de ejemplo, a veces en diferentes lenguajes como demostración.
* Usos conocidos, dando ejemplos de sistemas reales que lo aplican.
* Patrones relacionados, indicando los nombres de otros patrones con los que presenta similitudes o con los que es posible combinarlo.

El más conocido y antiguo de estos catálogos es el de **Gamma, Helm, Johnson y Vlissides** (patrones de Gamma), que publicaron una primera versión de su libro, Design Patterns, en 1994 y desde entonces han aparecido múltiples ediciones (Gamma, Helm, Johnson y Vlissides, 2003). El trabajo de estos cuatro autores ha sido tan influyente que generalmente son conocidos como «La banda de los cuatro» (Gang of Four [GoF]).

En su libro, Gamma et al. (2003) identifican **trece patrones de diseño** agrupados en tres categorías fundamentales:

* **Creacionales**: ayudan en la tarea de construcción de nuevos objetos de características complejas. Se encargan de la creación, composición y representación de objetos
  + Abstract factory (fabricación abstracta).
  + Builder(constructor).
  + Prototype (prototipo).
  + Singleton**.**
* **Estructurales:** centrados en la manera de combinar entre sí objetos que se agrupan en estructuras lógicas más complejas. Relacionados con el modo en el que se organizan e integran las clases y objetos para construir una estructura más grande. Separan la interfaz de la implementación. Nos aseguran independencia entre las capas software que se van creando.
  + Adapter(adaptador).
  + Composite(compuesto).
  + Facade(fachada).
  + Proxy.
* **De comportamiento:** ayudan a resolver problemas de comunicación entre objetos, proponiendo algoritmos adecuados. Enfocados a la asignación de responsabilidad entre los objetos y al modo en el que se comunican. Más que describir objetos o clases, describen la comunicación entre ellos.
  + Chain of responsibility (cadena de responsabilidad).
  + Mediator (mediador).
  + Observer (observador).
  + State (estado).
  + Template method (método plantilla).

### **Abstract Factory**

**Problema:** queremos trabajar con familias de objetos relacionados o dependientes entre sí, pero no nos preocupa la clase concreta, sino la manera de manipularlos.

Este patrón permite crear una interfaz de manipulación común para un conjunto de objetos. No interesan las características concretas de cada objeto, al menos desde el punto de vista de su utilización.

Pensemos en una aplicación con interfaz gráfica, que podemos generar empleando dos posibles alternativas de **familias de controles gráficos,** Tipo1 y Tipo2 (cada una de ellas compuesta por botones, listas desplegables, etc.). El mismo tipo de control dentro de cada familia se utiliza empleando el mismo conjunto de métodos. El diagrama mostrado en la Figura 16 permite resolver este tipo de problema. Al cliente que crea y manipula los controles no le interesa el tipo concreto de botón que se instancia, sino las **interfaces** que ofrece para su manipulación. La clase encargada de crear el objeto del tipo correcto en cada caso es una **factoría** que, a su vez, implementa unos métodos de creación conocidos por el cliente, independientemente del tipo de factoría concreta que estemos utilizando.

A diagram of a computer

Description automatically generated

Figura 16. Ejemplo de utilización del patrón de fábrica abstracta. Fuente: elaboración propia.

Una aplicación que manipula diferentes tipos de documentos o un videojuego con varios tipos de personajes son otros ejemplos de aplicaciones que podrían beneficiarse de este patrón, siempre y cuando estas familias de objetos puedan**compartir una interfaz de utilización común,** de manera que no importe la clase concreta que estemos empleando en cada caso.

### **Builder**

**Problema:** deseamos separar la construcción de un objeto (el conjunto de pasos necesarios) de su representación (la información que contiene), de manera que un mismo proceso de construcción pueda dar lugar a representaciones diferentes.

El patrón builder nos ayuda a encapsular el procedimiento de construcción de objetos complejos en una clase especializada. El cliente que desea crear el objeto simplemente utiliza los métodos ofrecidos por el constructor, que devuelve una instancia concreta con las características deseadas.

En la Figura 17 se encuentra una posible manera de modelar este patrón. De manera más concreta, supongamos que estamos desarrollando una aplicación capaz de efectuar conversión de vídeo. El resultado puede ser ofrecido en diferentes formatos contendores (AVI, MPG, MOV, etc.), siendo posible a la vez emplear diferentes códecs para la compresión de audio y vídeo, y realizar otro tipo de operaciones durante el procesamiento (recortado, escalado, etc.).

En este caso, podríamos implementar un **constructor concreto** para cada uno de los tipos de contenedor soportados, devolviendo a cada uno de ellos un conversor específico para ese formato. La configuración de cada uno de los conversores puede ser similar, su construcción requiere de los mismos pasos (selección de códec de vídeo, selección de códec de audio, etc.), de manera que el proceso de construcción, o al menos los pasos necesarios, quedan modelados en la clase abstracta «Constructor», de la que todos los constructores concretos heredan.

A diagram of a construction site

Description automatically generated

Figura 17. Diagrama de clases para un patrón constructor. Fuente: elaboración propia.

Un ejemplo claro de utilización de este patrón lo encontramos en el proceso de construcción de una **URI en código Java,** como se muestra en la Figura 18.

A computer screen with text

Description automatically generated

Figura 18. Ejemplo de utilización del patrón builder en código Java. Fuente: elaboración propia.

### **Prototype**

**Problema:** necesitamos crear en tiempo de ejecución nuevas instancias de objetos similares o idénticos a otros que ya existen en el programa.

Este patrón permite crear copias de otros objetos en tiempo de ejecución, de manera que las copias sean idénticas o con ciertas modificaciones.

La instancia prototípica implementa una interfaz que contiene el método , de manera que el cliente sabe que es posible obtener copias de manera sencilla de ese objeto. La Figura 19 muestra un esquema de esta **solución.**

### **Singleton**

**Problema:** necesitamos garantizar que en todo momento solo existe una instancia de un determinado tipo de objeto.

Este patrón define una clase con un único punto de acceso global, y es la propia clase la que garantiza que el objeto que devuelve es la única instancia de esa clase.

La **solución** consiste en ocultar el constructor de la clase y definir un método estático que devuelve la única instancia existente, como vemos en la Figura 20.

A screen shot of a computer program

Description automatically generated

Figura 20. Modelo de clase y código de ejemplo Java del patrón singleton.Fuente: elaboración propia.

### **Adapter**

**Problema:** necesitamos adaptar la interfaz que ofrece un componente para que los clientes puedan utilizarla. Este es un problema habitual cuando empleamos un **enfoque de reutilización de** software.Puede que necesitemos comunicar entre sí clases que ofrecen servicios con otras que pueden consumirlos, pero las interfaces de estos objetos son incompatibles.

La **solución,** que permite la colaboración, consiste en intercalar entre ambas una nueva clase que hace el trabajo de traducción.

Pensemos en un editor de dibujo que utiliza una librería que permite la creación de objetos gráficos. Las clases propias del dominio de aplicación aparecen representadas en amarillo en la Figura 21, mientras que las clases que proceden de la librería reutilizada se muestran en azul.

A diagram of a computer

Description automatically generated

Figura 21. Ejemplo de diagrama de clases para el patrón adaptador. Fuente: elaboración propia.

Otro ejemplo podría ser el de una aplicación que consume información de diferentes API, todas ellas con un tipo de información semejante (por ejemplo, comentarios de usuarios en noticias de diferentes medios especializados en un tema), pero que desde el punto de vista de la aplicación es necesario uniformizar su uso estableciendo una nomenclatura de variables uniforme.

### **Composite**

**Problema:** tenemos un conjunto de objetos similares que se agrupan jerárquicamente, siendo unos contenedores de otros, y queremos uniformizar y simplificar el trabajo con los diferentes elementos, individuales o compuestos.

Hay aplicaciones que, de manera natural, requieren de la **agrupación recursiva de elementos.** Por ejemplo, un editor de dibujo que permite agrupar un conjunto de formas básicas, o grupos de ellas, entre sí o una aplicación que muestra una conversación entre usuarios con respuestas anidadas.

La **solución** consiste en definir «una clase abstracta que representa tanto a primitivas como a sus contendores» (Gamma et al., 2003, p. 151). Esta idea se muestra en el diagrama de la Figura 22.

A diagram of a computer

Description automatically generated

Figura 22. Ejemplo de diseño para el patrón composite. Fuente: elaboración propia.

### **Facade**

**Problema:** nuestros objetos necesitan interactuar con un conjunto de interfaces de objetos de un subsistema, lo cual lleva a un elevado acoplamiento.

El patrón de fachada proporciona una interfaz unificada y un punto de acceso común para un conjunto de interfaces de un sistema, simplificando su empleo.

Pensemos en una aplicación con interfaz de usuario en la que existen varias ventanas, diferentes en la capa de presentación y diferentes clases en la capa de negocio, especializadas en operaciones concretas. Puede ser útil insertar una capa intermedia entre la capa de presentación y la de negocio, o fachada, que ofrezca un único punto de acceso a las funcionalidades ofrecidas. De esta manera, todas las peticiones son gestionadas por la fachada, que luego las encamina al objeto de dominio apropiado. También es una manera de simplificar la utilización de un conjunto de objetos que ofrecen servicios, y de disminuir el acoplamiento entre nuestras clases y las del sistema que ofrece los servicios que estamos consumiendo.

A diagram of a service

Description automatically generated

Figura 23. Diagrama de ejemplo para el patrón de fachada. A la izquierda se muestra el problema original, y a la derecha la solución que ofrece el patrón. Fuente: elaboración propia.

### **Proxy**

**Problema:** necesitamos acceder a operaciones de un objeto, pero queremos añadir alguna función extra, o el acceso directo resulta demasiado complejo o costoso.

La **solución** consiste en definir un objeto proxy (o apoderado) que es un representante o sustituto del objeto real (al que llamaremos sujeto), permitiendo su manipulación indirecta. Se consigue de esta manera un mecanismo más sofisticado de manipulación de un objeto que permite agregar funcionalidad.

En muchos casos, el objeto sustituido es un sistema o un dispositivo externo, siendo habitual su empleo en **operaciones de acceso a la red.** En este caso, el patrón de fachada es equivalente al patrón proxy.

Otra aplicación habitual es la sustitución de un objeto cuya carga en memoria puede resultar**muy lenta o costosa.** Por ejemplo, en un procesador de textos cada uno de los elementos del documento está asociado a un objeto que existe en la memoria del computador. Interesa que la carga de los documentos sea rápida, pero algunos elementos, como las imágenes, pueden resultar voluminosos, ocupar gran espacio en memoria y ralentizar la apertura del documento. En este caso, se pueden sustituir las imágenes por objetos proxy en el momento de la carga del documento, de manera que se aplaza la carga de la imagen del disco a memoria hasta el momento en el que debe ser representada al usuario en la pantalla.

Encontramos, así, **diferentes tipos de proxy** (Gamma et al., 2003, p. 192):

* **Remoto:** que proporciona un representante local para un objeto que se encuentra en una máquina diferente.
* **Virtual:** que se encarga de la creación de objetos más complejos bajo demanda.
* **De protección:** que establece controles adicionales de acceso a los métodos ofrecidos por el sujeto real.
* **Inteligente:** que sustituye a un puntero real realizando operaciones adicionales cada vez que el objeto es manipulado. Por ejemplo, puede llevar una contabilidad del número de referencias al objeto real, de manera que la memoria que ocupa pueda ser liberada cuando el objeto ya no es necesario.

En las Figuras 24 y 25 se muestra un diseño simplificado de este tipo de **solución** para un problema genérico. Tanto el objeto proxy como el sujeto real implementan la misma interfaz. Sin embargo, el cliente únicamente colabora con el objeto proxy, que es el único que conoce al sujeto real que ofrece la funcionalidad solicitada.

A diagram of a service

Description automatically generated

Figura 24. Diagrama de clases como ejemplo de un patrón proxy. Fuente: elaboración propia.

A diagram of a block diagram

Description automatically generated

Figura 25. Diagrama de secuencia como ejemplo de un patrón proxy. Fuente: elaboración propia.

### **Chain of responsibility**

**Problema:** se genera un evento en el sistema que puede ser procesado por uno o varios objetos diferentes y se busca la mejor manera de dirigir el flujo de procesamiento del evento.

Este patrón establece una cadena de objetos que pasan de unos a otros una petición hasta que se encuentra un objeto capaz de procesarla. De esta manera, el emisor y el receptor de la petición quedan desacoplados.

Una **solución** consiste en definir una clase abstracta con un método capaz de responder a esa petición y un objeto sucesor al cual pasar la petición en caso de ser necesario. Todas las instancias concretas implementan esta interfaz, las que pueden responder a la petición lo hacen y, en caso contrario, la derivan al sucesor asignado.

Supongamos que un procesador de texto ofrece un sistema de ayuda contextual al usuario simplemente pulsando sobre un elemento de la interfaz. La ayuda ofrecida debería ser diferente en función del elemento seleccionado. Puede que algunos elementos no ofrezcan un mensaje de ayuda concreto, pero en este caso sería bueno que delegaran esta funcionalidad en algún otro objeto de jerarquía superior que sea capaz de procesar el evento, buscando siempre una información de ayuda de lo más específico a lo más general, como se representa en la Figura 26.

A screenshot of a computer

Description automatically generated

Figura 26. Ejemplo de patrón de cadena de responsabilidad en un procesador de textos. Fuente: elaboración propia.

### **Mediator**

**Problema:** existe un conjunto de objetos que colaboran habitualmente entre sí intercambiando información o invocando métodos ajenos y se desea reducir el acoplamiento entre ellos.

En muchos sistemas encontramos conjuntos de objetos que deben colaborar o comunicarse entre sí. Una **solución** simple podría conducir a que todos los objetos conocieran a todos los restantes, aumentando innecesariamente el número de conexiones y, por tanto, el acoplamiento general del sistema.

La solución de este patrón consiste en definir un objeto mediador que es «responsable de controlar y coordinar las interacciones entre un grupo de objetos» (Gamma et al., 2003, p. 252).

Pensemos en una interfaz de usuario en la que modificaciones en el estado de algún componente afectan a otros (por ejemplo, si un campo de texto está vacío se deshabilita un botón o si se supera un número de caracteres se muestra un mensaje de alerta). Podría ser una ventana de login a un sistema, como se muestra en la Figura 27.

A diagram of a diagram

Description automatically generated

Figura 27. Ejemplo de patrón mediador en una ventana de login a un sistema. A la izquierda encontramos un sistema fuertemente acoplado, con muchas dependencias entre sus clases. El patrón mediador empleado en el ejemplo de la izquierda reduce el acoplamiento, introduciendo un objeto central de coordinación de la comunicación entre el resto de los participantes. Fuente: elaboración propia.

El estado de los diferentes elementos de entrada de información puede afectar a la habilitación o deshabilitación de otros elementos de la interfaz. Mientras que a la izquierda de la figura tenemos un conjunto de objetos muy acoplados entre sí, la solución propuesta, introduciendo un mediador, reduce el acoplamiento y facilita la mantenibilidad.

### **Observer**

**Problema:** un conjunto de objetos debe ser notificado acerca de los cambios que se producen en otro objeto del sistema.

El patrón que resuelve este problema es similar al mediador, pero en este caso se trata únicamente de notificar sobre los cambios en el estado de un objeto a otro conjunto de objetos.

La **solución** consiste en definir dos tipos de objetos fundamentales:

* El **sujeto,** donde se producen los cambios en la información.
* El **observador,** que es notificado por el sujeto sobre estos cambios.

Se define así un modelo de **publicación-suscripción,** en el que el sujeto notifica a sus observadores cuando se produce un cambio, como se muestra en el ejemplo de la Figura 28.

A diagram of a computer

Description automatically generated with medium confidence

Figura 28. Ejemplo de utilización del patrón observador. Fuente: elaboración propia.

La secuencia de interacciones se representa en el diagrama de la Figura 29.

A diagram of a project

Description automatically generated

Figura 29. Diagrama de secuencia mostrando las interacciones entre clases empleando un patrón observador. Fuente: elaboración propia.

Como podemos ver, el sujeto mantiene una lista de todos los observadores que se han registrado con él. Cuando se produce un cambio en su estado interno lo notifica a todos ellos, de manera que puedan sincronizarse y adquirir la nueva información de estado.

### **State**

**Problema:** el comportamiento de un objeto depende de su estado interno.

En muchas aplicaciones el comportamiento de un objeto viene afectado por su estado interno o depende de ciertas variables. En estos casos, es posible delegar la implementación de esa funcionalidad correspondiente en una clase asociada.

Supongamos, por ejemplo, un videojuego en el que el comportamiento de un personaje depende de ciertas variables, como su nivel de energía, salud, o experiencia. En este caso, el personaje es un contexto que mantiene una instancia de una clase concreta que representa su estado actual (ver Figura 30). El contexto ofrece una interfaz que permite a sus clientes realizar peticiones pero, cuando estas llegan, el contexto ejecuta el método de gestión de la instancia concreta, que representa el estado actual, y es capaz de responder a esa petición.

A diagram of a diagram

Description automatically generated

Figura 30. Ejemplo de utilización del patrón de estado. Fuente: elaboración propia.

### **Template method**

**Problema:** existe un conjunto de objetos que realizan un tipo de operaciones similares, con una estructura algorítmica general similar, pero se diferencian en algunos pasos del proceso. Se desea construir una jerarquía de clases que evite la duplicación de código.

La **solución** consiste en crear en una clase base un método que implementa el esqueleto del algoritmo (método plantilla), haciendo uso de métodos abstractos que deberán ser concretados en las clases derivadas, donde se puede agregar funcionalidad específica. De esta manera, la secuencia de pasos lógicos necesarios queda fija y solo es preciso agregar el comportamiento específico en las clases concretas que utiliza la aplicación.

En la Figura 31 vemos un ejemplo de ello. Supongamos que estamos desarrollando un visor de documentos que soporta varios formatos de archivos y tipos de documentos (hoja de cálculo, imagen, documentos de texto, etc.). El proceso de carga es básicamente el mismo en cuanto al procedimiento, pero se diferencia en función del tipo de documento específico que debe ser cargado y representado de manera diferente. En la clase base se define el método plantilla , que define los pasos necesarios para cargar y representar un documento cuyo nombre es recibido como parámetro. Se encarga de realizar la comprobación de que el formato es soportado empleando un método virtual, , y a partir de ahí define la secuencia de pasos necesarios para mostrar finalmente el contenido del archivo, como vemos en la Figura 32.

A diagram of a document

Description automatically generated

Figura 31. Ejemplo de aplicación del patrón de método plantilla. Fuente: elaboración propia.

A computer screen shot of a program code

Description automatically generated

Figura 32. Código de ejemplo de implementación de un patrón de método plantil