Tema 4

Desarrollo de software basado en reutilización

Ingeniería del Software Avanzada

Índice

[Esquema 3](#_Toc532285625)

[Ideas clave 4](#_Toc532285626)

[4.1. Introducción y objetivos 4](#_Toc532285627)

[4.2. La reutilización del software 5](#_Toc532285628)

[4.3. Niveles de reutilización de código 6](#_Toc532285629)

[4.4. Técnicas concretas de reutilización 7](#_Toc532285630)

[4.5. Desarrollo basado en componentes 18](#_Toc532285631)

[4.6. Beneficios y problemas de la reutilización 22](#_Toc532285632)

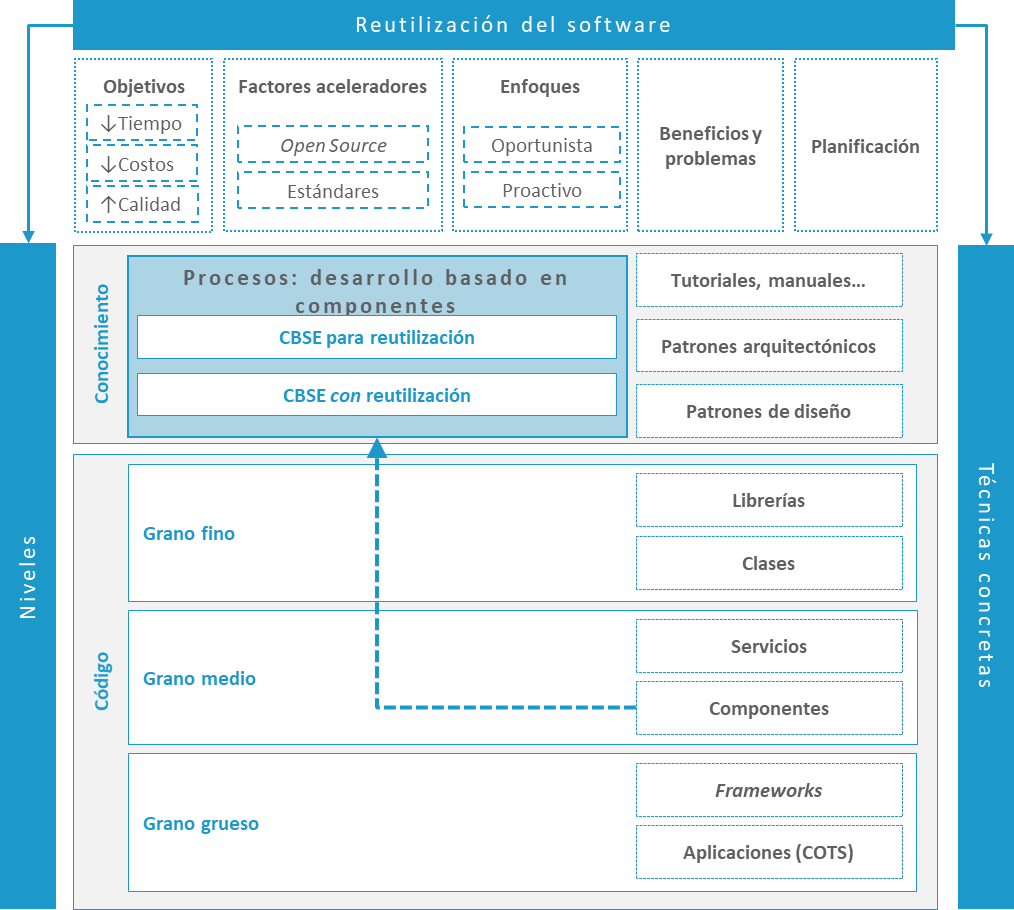
[4.7. Planificación de la reutilización 24](#_Toc532285633)

[4.8. Referencias bibliográficas 25](#_Toc532285634)

[A fondo 27](#_Toc532285635)

[Test 29](#_Toc532285636)

Esquema



Ideas clave

4.1. Introducción y objetivos

En este tema estudiaremos la manera de **abordar el proceso de desarrollo de software basado en la reutilización**. Se trata de una filosofía o estrategia que guía el proceso de ingeniería para **aprovechar código y conocimientos generados previamente**.

Comenzaremos analizando las principales **características y objetivos** de la reutilización del software y, a continuación, explicaremos **técnicas específicas** relacionadas con este concepto. También estudiaremos las **particularidades del proceso de software basado en componentes** y las principales consideraciones que debemos tener en cuenta al adoptar este enfoque general.

Los **objetivos** que perseguimos alcanzar son los siguientes:

* **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 y **saber agruparlas** según su *granularidad* o cantidad de código reutilizado.
* **Saber relacionar la importancia de los patrones arquitectónicos y de diseño** con otras técnicas de reutilización de software.
* Conocer las **actividades fundamentales del proceso de software basado en componentes**.
* **Conocer los aspectos a valorar** antes de decidir por la utilización de estas técnicas.

4.2. La reutilización del software

El **desarrollo orientado a la reutilización**, aunque conceptualmente data de mucho tiempo atrás, 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:

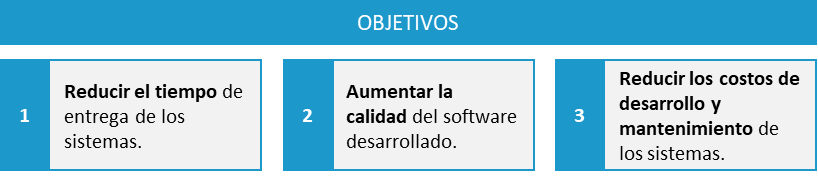


Figura 1. Objetivos de desarrollo del software

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

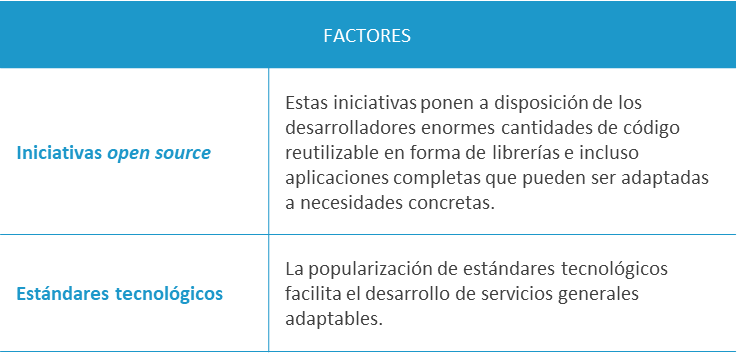


Figura 2. Factores de impulso del desarrollo orientado a la reutilización

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):

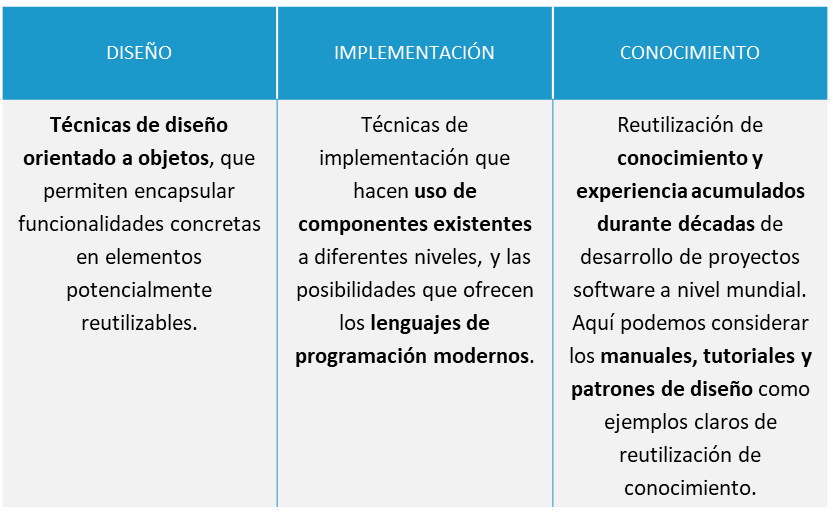


Figura 3. Conceptos sobre los que se apoya la reutilización de software

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):

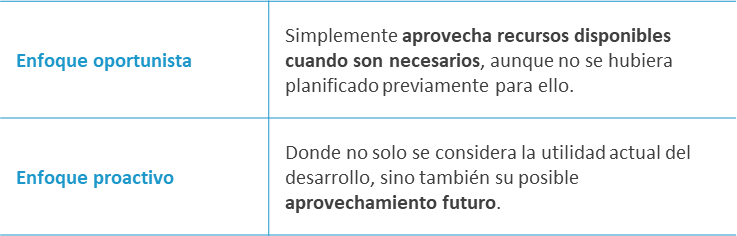


Figura 4. Tipos de enfoque adoptados en el proceso de reutilización

4.3. 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):

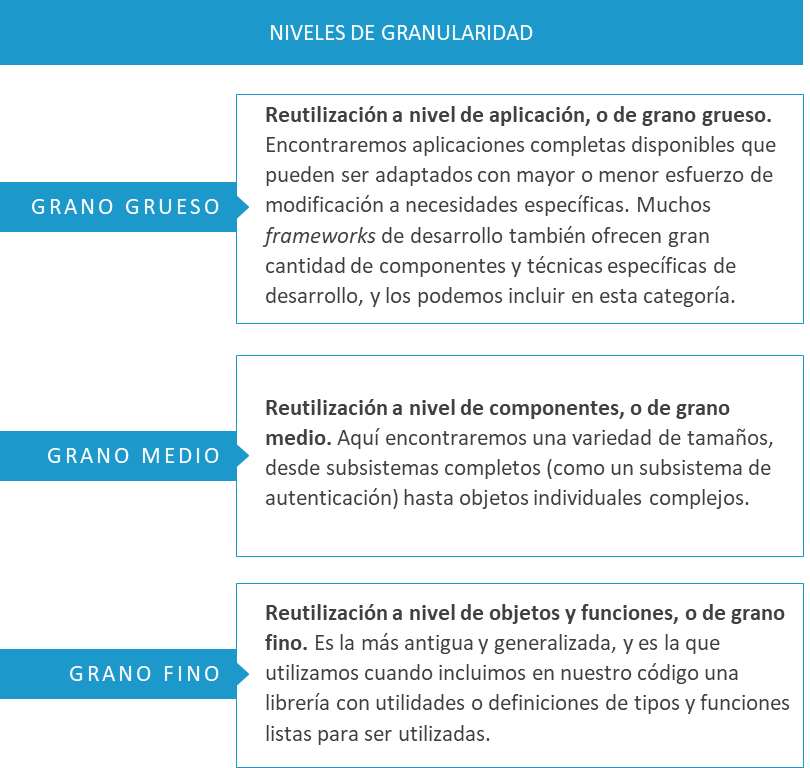


Figura 5. Niveles de granularidad en función de la cantidad y complejidad del código reutilizado

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

4.4. Técnicas concretas de reutilización

En este apartado abordaremos algunas **soluciones técnicas concretas** que permiten la reutilización de software.

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 permite **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 de un solo golpe de vista.
* A lo largo de los años se han producido **multitud de patrones arquitectónicos** que dan respuesta a problemas concretos. En el próximo tema analizaremos en profundidad algunos de ellos.

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 diseño consideran intrínsecamente la reutilización como un valor deseable**, de manera que cuando aplicamos un patrón no solamente estamos reutilizando un diseño, sino a su vez el patrón definirá ciertos elementos (clases) reutilizables en nuestra aplicación. Esto, en ocasiones, puede requerir 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.

Sobre estos temas profundizaremos mucho más en próximos temas.

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**:

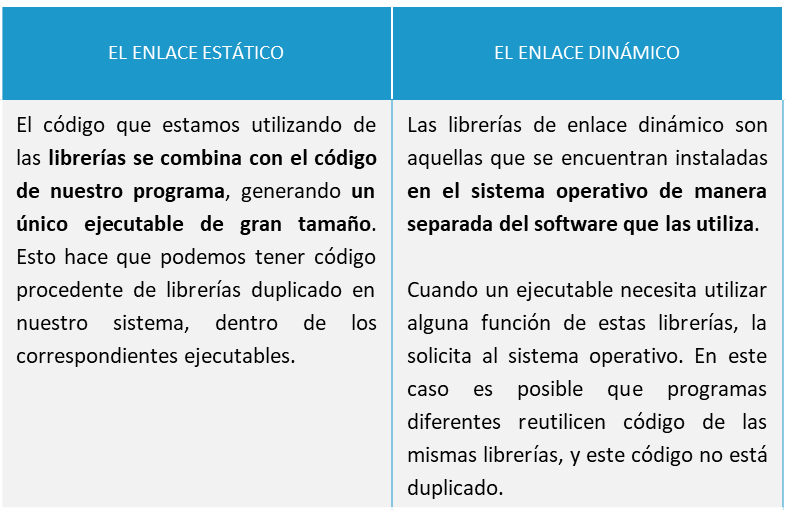


Figura 6. Enfoques de reutilización del código de las librerías

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 venido refiriendo 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

La aparición del paradigma de la programación orientada a objetos (POO) y las capacidades —como la herencia, el polimorfismo y la encapsulación de funcionalidades— están directamente relacionadas con la búsqueda de la reutilización de código.

**Generalmente, las clases se agrupan en paquetes que contienen clases relacionadas entre sí.**

Veamos cómo **algunos de los conceptos de la POO ayudan en la reutilización de código:**

* La **abstracción** hace referencia a la posibilidad de extraer las características esenciales y relevantes de objetos dentro de una categoría que los diferencian del resto. Desde el punto de vista externo, solo nos interesan los servicios que ofrece un objeto y no tanto su funcionamiento interno.
* El **encapsulamiento** es lo que permite encerrar en el objeto tanto la información sobre su estado (en campos o atributos) como su comportamiento (en operaciones o métodos). No debe confundirse con el concepto de ocultación, aunque, generalmente, los atributos permanecen ocultos al exterior y el objeto solo expone aquellos servicios que desea ofrecer a través de métodos públicos, siendo el resto invisibles (e innecesarios) para el consumidor externo.
* La **herencia** permite crear jerarquías completas de clases relacionadas, de manera que las descendientes heredan comportamientos de los padres, no siendo necesario repetir el código que describe una funcionalidad común.
* El **polimorfismo** permite trabajar con diferentes objetos sin importar su tipo concreto, siempre que soporten una interfaz definida y esperada por el consumidor. Esto quiere decir que la invocación de un mismo método en distintos objetos puede provocar comportamientos diferentes.

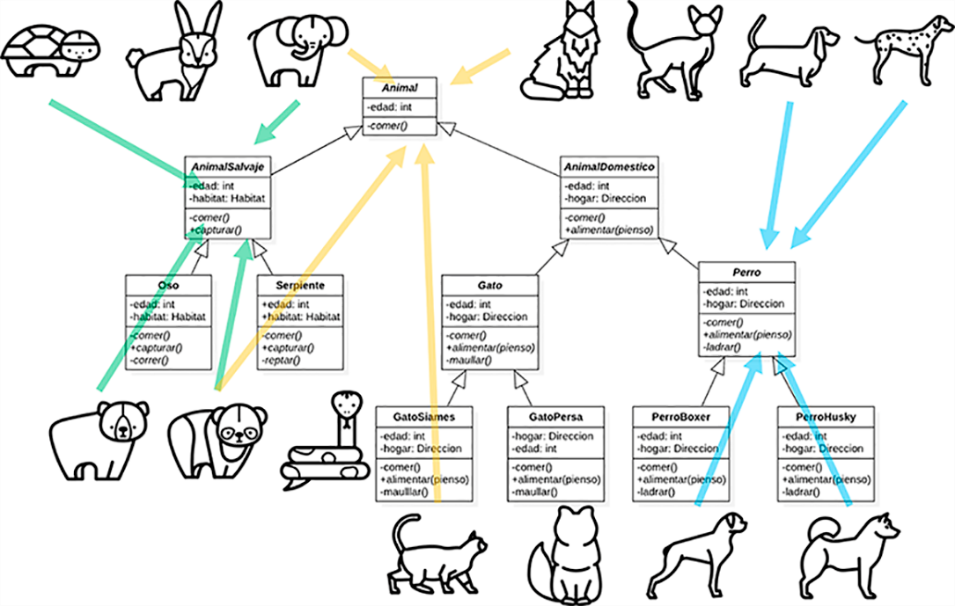


Figura 7. El paradigma de la programación orientada a objetos

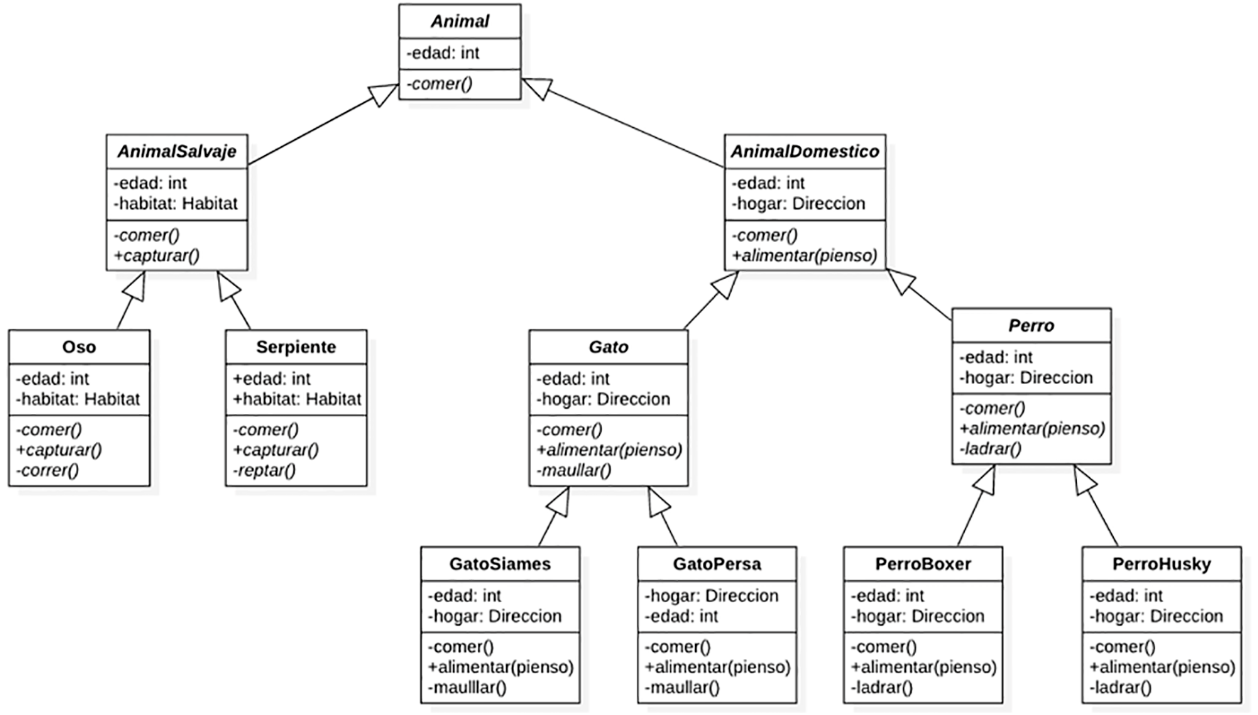


Figura 8. Detalle de la figura 7

Todas estas ideas aparecen representadas en la figura 7. Todos los animales, sin importar su tipo, comparten características comunes que se abstraen en la clase **Animal** (las flechas naranjas indican esta relación de abstracción). Por otro lado, los comportamientos y atributos compartidos por todos los perros se encapsulan en la clase abstracta **Perros** (flechas naranjas), y las características de los animales salvajes se modelan en la clase **AnimalSalvaje** (flechas verdes). Vemos así que las variables y métodos propios de cada clase quedan encapsuladas de manera independiente, y que los objetos solo hacen públicas aquellas operaciones que pueden resultar útiles desde el exterior (alimentar() con un pienso, o fotografiar() a un animal).

Finalmente, desde el punto de vista del consumidor del servicio puede ser irrelevante la instancia concreta con la que se esté trabajando. Un objeto Fotógrafo sabe que cualquier AnimalSalvaje expone públicamente el método fotografiar(), y el comportamiento que desencadene su invocación dependerá únicamente de la instancia concreta con la que se esté trabajando.

Los servicios

**Un servicio es, según Polo Usaola (2012) «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 en 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).**

El desarrollo basado en componentes se ocupa de desarrollar sistemas integrando componentes, y da lugar a un modelo de proceso especializado que estudiaremos más adelante.

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 9):

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

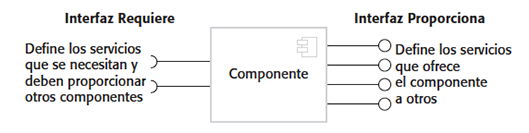


Figura 9. Interfaces de un componente. La interfaz «proporciona» es siempre necesaria

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, Gokhale y Natarajan (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):

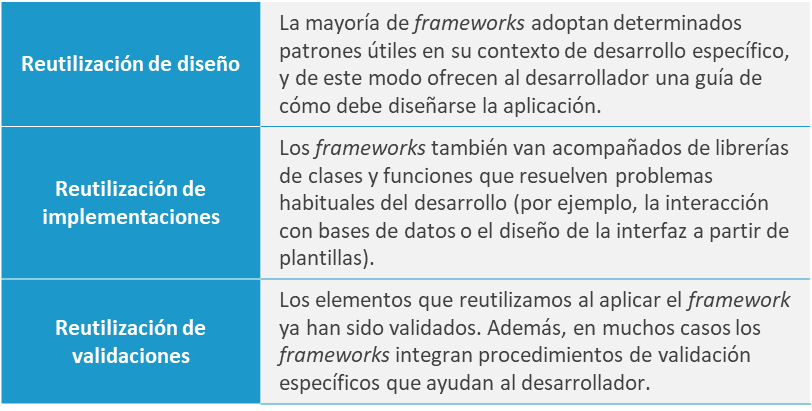


Figura 10. Cómo mejoran los *frameworks* la extensibilidad y portabilidad del sistema

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 llegar 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-end* que permiten crear páginas más interactivas.

Reutilización de aplicaciones (productos COTS)

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.

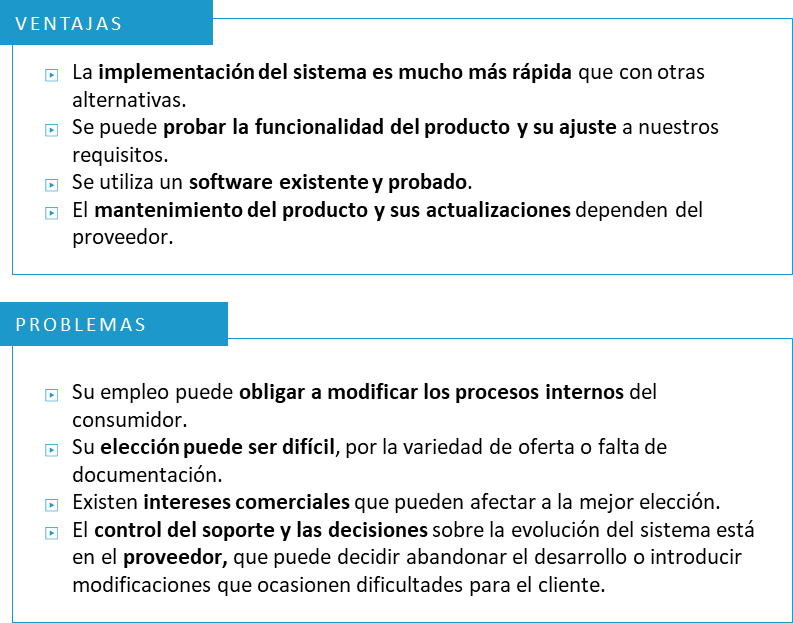


Figura 11. Ventajas y problemas de los productos COTS

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

|  |  |
| --- | --- |
| **SISTEMAS DE SOLUCIÓN COTS** | **SISTEMAS INTEGRADOS COTS** |
| Un único producto satisface los requisitos. | Se integran varios sistemas de solución. |
| Solución genérica basada en procesos estándar. | Mayor flexibilidad de adaptación a los procesos del cliente. |
| El desarrollo es la configuración del sistema. | El desarrollo es la integración del sistema. |
| Proveedor responsable del mantenimiento. | Cliente es responsable del mantenimiento. |
| El proveedor aporta la plataforma. | El cliente debe aportar la plataforma para el sistema. |

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

**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.
    - 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 12 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 estudiaremos a continuación (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.

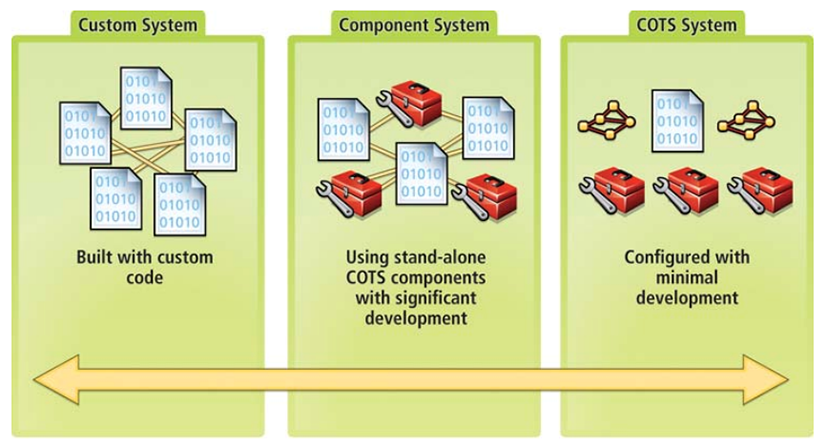


Figura 12. Diferencias entre el desarrollo de un sistema a medida, su construcción a partir de componentes y la utilización de un sistema COTS. Fuente: Esri (2011)

4.5. Desarrollo basado en componentes

**Como hemos visto, un componente software es un fragmento de código que encapsula un conjunto de funcionalidades específicas, y relacionadas entre sí, que se exponen a través de interfaces estandarizadas, de manera que se facilita su reutilización. Su combinación adecuada permite crear aplicaciones más completas.**

La labor de ingeniería, que analiza los sistemas aprovechando este enfoque de reutilización, define una arquitectura de colaboración e implementa el código necesario para que los componentes funcionen unos con otros, es conocida como **desarrollo basado en componentes, o ingeniería del software basada en componentes** —*Component Based Software Engineering* (CBSE)—.

Existen **dos tipos de procesos CBSE**:

1. **Desarrollo «para» reutilización**. Se centra en el desarrollo de componentes que podrán ser reutilizados y que, en general, supone generalizar componentes existentes.
2. **Desarrollo «con» reutilización**. Busca desarrollar aplicaciones nuevas empleando componentes y servicios existentes.

CBSE para reutilización

Los primeros defensores del CBSE pronosticaban que era posible desarrollar un nuevo e importante mercado de componentes, existiendo empresas especializadas en su desarrollo y vendedores que los proporcionarían a otros desarrolladores. Sin embargo, **actualmente existen pocos proveedores de componentes y su compra es poco habitual** (seguramente debido en parte a los problemas que más adelante veremos).

Si consideramos un servicio externo como un componente más del sistema, sí es cierto que gracias a la computación en la «nube» el mercado se ha venido desarrollando de una manera mucho más intensa.

Sí es muy habitual que este enfoque de desarrollo de componentes orientado a su futura reutilización se realice dentro de las organizaciones. Para ello, es necesario en general adaptar desarrollos existentes teniendo en cuenta los siguientes cambios necesario (Polo Usaola, 2012; Sommerville, 2011):

* **Eliminar dependencias** de métodos específicos del sistema para el que fue desarrollado.
* **Renombrar los servicios** ofrecidos para hacerlos más generales e independientes del dominio inicial.
* **Agregar nuevos servicios** para cubrir un campo de necesidades más amplio.
* **Ofrecer un conjunto de excepciones** para cada uno de los servicios ofertados, de manera que el consumidor tenga información sobre el posible comportamiento anómalo del componente.
* **Incluir una interfaz de configuración** que permita su adaptación a necesidades específicas.
* **Incluir posibles dependencias** en un componente integrado, de manera que el usuario del componente vea facilitada su integración.

CBSE con reutilización

El desarrollo de nuevos sistemas software como integración de componentes existentes requiere de una adaptación del proceso general de desarrollo.

Debemos tener en cuenta que los componentes deben ser considerados como «cajas negras», de manera que deben ser utilizados tal y como son proporcionados, y su modificación es a menudo complicada, imposible o no conveniente. Este modelo se representa de manera general en la figura 13.

Las adaptaciones necesarias frente a otros modelos de proceso son las siguientes:

* **Especificación inicial**: se parte de una especificación de requisitos genérica, ya que puede ser necesaria su adaptación en función de las alternativas de componentes encontradas. Una especificación demasiado detallada puede limitar el número de alternativas.
* **Adaptación de requisitos**: se adaptan los requisitos en función de los componentes encontrados. Para cada componente se evalúa:
  + Su capacidad para cumplir con los requisitos definidos.
  + Su encaje en el diseño arquitectónico del sistema.
  + La necesidad de adaptación, analizando sus interfaces, dependencias, documentación, seguridad y cualquier factor que afecte a su integración.
* **Selección e integración**: una vez seleccionados los componentes, se procede a su composición e integración en el sistema, que puede requerir de adaptaciones.
* **Evolución:** finalmente, hay que considerar la fase de mantenimiento. Puede ser necesario:
  + Corregir errores: **mantenimiento correctivo**.
  + Agregar funcionalidades: **mantenimiento perfectivo**.
  + Adaptar el sistema a cambios en el entorno: **mantenimiento adaptativo**.

Estas modificaciones pueden suponer la sustitución de componentes por otros o por nuevas versiones de estos, o su modificación interna en caso de que sea posible.

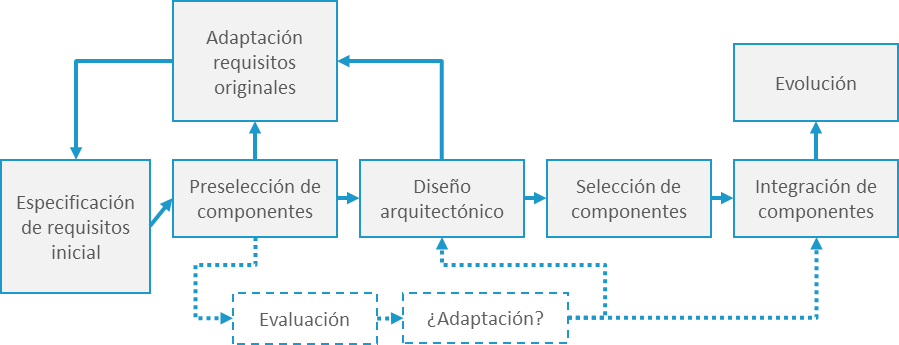


Figura 13. Modelo general de proceso CBSE con reutilización

Composición de componentes

**La composición de componentes implica combinarlos entre sí y desarrollar el código necesario para que el sistema quede integrado.**

Esta integración se puede realizar de **tres maneras diferentes** (Sommerville, 2011), como vemos en la figura 14:

* **Composición secuencial**. Se crea un sistema compuesto utilizando secuencialmente los servicios ofrecidos por A, y utilizando estos resultados como entrada para los servicios que ofrece B. En este caso, puede ser necesario código adicional para establecer el orden correcto de las llamadas y garantizar que las salidas de A son compatibles con las entradas que espera B.
* **Composición jerárquica**. En este caso, dos componentes se conectan directamente entre sí, de manera que es necesario que la interfaz «requiere» de A sea compatible con la interfaz «proporciona» de B. Puede ser necesario realizar una adaptación entre ambas interfaces.
* **Composición aditiva**. Se crea un nuevo componente virtual como combinación de dos o más componentes existentes. En conjunto, ofrecen nuevas interfaces «proporciona» y «requiere» que son combinación de las de sus integrantes y, generalmente, es necesaria la creación de una fachada que uniformice su uso.

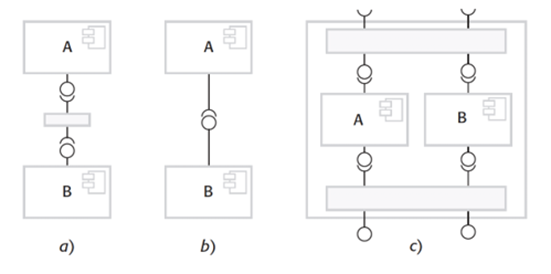


Figura 14. Diferentes tipos de composición de componentes: secuencial (a), jerárquica (b) y aditiva (c)

Como vemos, la integración de componentes puede requerir el desarrollo de algún mecanismo de adaptación, esto se debe a **varios posibles tipos de incompatibilidad:**

* **Incompatibilidad de parámetros**. Las operaciones en ambas interfaces conectadas tienen el mismo nombre, pero distinta organización o tipos de parámetros.
* **Incompatibilidad de operaciones**. Los nombres de las operaciones en ambas interfaces son incompatibles.
* ***Incompletitud* de operaciones**. La interfaz «proporciona» de un componente es insuficiente para la interfaz «requiere» del otro componente.

4.6. Beneficios y problemas de la reutilización

La reutilización de código y conocimiento acumulados en proyectos previos ofrece los siguientes **beneficios** en los procesos de desarrollo (Polo Usaola, 2012; Sommerville, 2011):

* **Disminución de la duración del proyecto y los plazos de entrega**, al aprovechar elementos ya existentes.
* **Reducción de costos del proyecto**, como consecuencia de lo anterior y de evitar el desarrollo de algunas partes del sistema ya disponibles.
* **Aumento de la confiabilidad en el sistema**, al reutilizarse componentes probados en desarrollos anteriores.
* **Disminución del riesgo del proyecto**, tanto desde el punto de vista tecnológico, como en la estimación de costos. El valor y calidad del software existente es ya conocido, mientras que a menudo es más complejo valorar el costo de su desarrollo.
* **Optimización del conocimiento especializado**. Por un lado, los especialistas en un tema pueden desarrollar componentes reutilizables que encapsulan su conocimiento en ese ámbito en concreto. Por otro, los desarrolladores de nuevos sistemas pueden reutilizar componentes genéricos o específicos, centrando su labor de manera más eficiente en aspectos relativos al dominio de aplicación concreto.
* **Mejora de la experiencia de usuario**. La reutilización de componentes de interfaz de usuario estandarizados permite homogeneizar la apariencia externa de una categoría de aplicaciones, aumentando la familiaridad para el usuario y facilitando el uso de las aplicaciones.
* **Mejora de la arquitectura y el diseño del sistema**. Primero porque este enfoque obliga a un proceso de análisis más detallado y, segundo, porque la reutilización de *frameworks* y librerías existentes imponen restricciones que generalmente conducen a un mejor estilo en la construcción del sistema.
* **Supone una inversión a futuro**, ya que los elementos con posibilidad de reutilización repercutirán favorablemente en desarrollos futuros.
* **Mejora del mantenimientodel sistema**, puesto que es más fácil detectar problemas en componentes que se utilizan en varios sistemas y, una vez arreglado el componente, se benefician todos los desarrollos que lo integran. En el caso de utilizar componentes de terceros, nos podemos beneficiar del esfuerzo realizado por esa comunidad de desarrolladores en la mejora de esos componentes.

A pesar de todo ello, también podemos identificar algunos aspectos posiblemente negativos en este enfoque. Enumeramos algunos de estos posibles **problemas**:

* **Problemas de mantenimiento**, si no tenemos acceso o conocimientos para modificar el software reutilizado.
* **La creación y mantenimiento de la librería de componentes** supone un costo adicional.
* **Costes de aprendizaje y comprensión de los componentes reutilizados**, al menos la primera vez que se utilizan.
* **Costes de integración en el sistema**, que puede requerir el desarrollo de adaptadores específicos.
* **Síndrome de «no se inventó aquí»,** Sommerville (2011, p. 428), que hace que muchos ingenieros prefieran escribir sus propios componentes, por considerar que lo harán mejor y que resulta una tarea más desafiante.

4.7. Planificación de la reutilización

Si adoptamos un enfoque proactivo en la reutilización del software, deberemos considerar una serie de factores relacionados con el desarrollo del proyecto (Sommerville, 2011) y, así, determinar la **técnica o el conjunto de técnicas más adecuadas para cada caso concreto**:

* **El tiempo disponible para el desarrollo**. Si la planificación presenta tiempos ajustados, conviene adoptar estrategias de mayor *granularidad*, de manera que se puedan desarrollar partes importantes del sistema, o el sistema completo, en plazos reducidos.
* **La vida esperada del software**. El mantenimiento futuro del software es un aspecto a tener en cuenta. Utilizar componentes disponibles puede facilitar el proceso de mantenimiento, si es que estos cuentan con su propia comunidad de desarrollo que introduce mejoras, arregla problemas, mejora la seguridad… Pero el no tener acceso al código de estos componentes puede dificultar el mantenimiento, si somos nosotros quienes queremos introducir la modificación.
* **La experiencia del equipo de desarrollo**. A menudo, la integración de un componente existente en un nuevo desarrollo puede ser compleja, o requerir de habilidades adicionales. Por otro lado, la reutilización de elementos genéricos puede permitir al equipo focalizarse en aquellos aspectos específicos del desarrollo en los que se tiene mayor experiencia.
* **La criticidad del software**. Si el desarrollo debe ser certificado por un regulador externo, o hay que tener en cuenta aspectos específicos relacionados con su seguridad o eficiencia; no tener acceso al código fuente o capacidad para modificarlo puede suponer un problema.
* **Los requisitos no funcionales**. El empleo de estrategias de reutilización, como la generación automática de código, puede suponer un detrimento de la eficiencia del sistema, y de otros aspectos no funcionales, que conviene evitar.
* **El dominio de aplicación**. La cantidad de software disponible para su reutilización puede no ser la misma dependiendo del dominio o aplicación concreta, y la adaptación de código existente puede no ser sencilla o viable en todos los casos.
* **La plataforma de despliegue del sistema**. Pueden existir problemas de compatibilidad entre la tecnología propia del proyecto y la de los componentes que se desea reutilizar. La adaptación o integración de todos ellos puede resultar demasiado compleja.

4.8. Referencias bibliográficas

del Pino, J. (2017). Frameworks Web de lado servidor [Web]. Recuperado de <https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros_pasos/Web_frameworks>

Esri. (2011). Taking a COTS-Based Approach to Implementing Enterprise GIS (An Esri White Paper). Recuperado de <http://www.esri.com/library/whitepapers/pdfs/cots-based-approach-enterprise.pdf>

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (2003). *Patrones de diseño: elementos de software orientado a objetos reutilizables*. Madrid: Pearson Educación.

Heineman, G. T. & Councill, W. T. (2001). *Component-based software engineering: putting the pieces together* (1.a ed.). Boston: Addison-Wesley.

Johnson, R. E. (1997). Frameworks = (components + patterns). *Communications of the ACM, 40*(10), 39-42. doi:10.1145/262793.262799

Polo, M. (2012). *Desarrollo de software basado en reutilización* [Material didáctico]. Barcelona: Universitat Oberta de Catalunya. Recuperado de <http://hdl.handle.net/10609/63466>

Schmidth, D. C., Gokhale, A. & Natarajan, B. (2004). Leveraging Application Frameworks. *ACM Queue, 2*(5), 66-75.

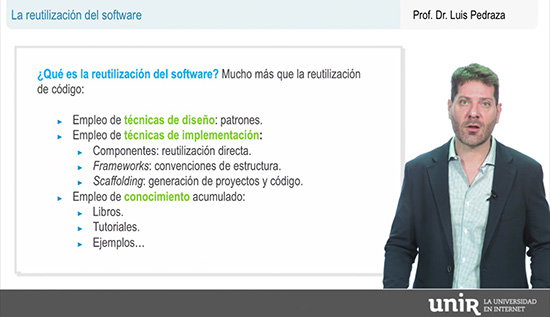
Sommerville, I. (2011). *Ingeniería de software* (9ª ed.). México: Pearson Educación de México.

W3C. (2004). Web Services Glossary [Web]. Recuperado de <https://www.w3.org/TR/ws-gloss/>

A fondo

**Lección magistral: La reutilización del software**

En esta lección magistral introduciremos el concepto de «reutilización del software» que busca, entre otros objetivos, reducir los tiempos de desarrollo del producto (reduciendo, por tanto, también el coste) y aumentar la calidad del sistema producido. Resaltaremos la importancia de integrar esta práctica de manera rutinaria en nuestros procesos de desarrollo, y revisaremos los diferentes niveles en los que se puede producir esta reutilización.



Accede a la lección magistral a través del aula virtual.

**Frameworks Web del lado servidor**

En este artículo se explora cómo los *frameworks* web pueden facilitar el desarrollo de aplicaciones web y se explican algunos de los criterios que se pueden usar para elegir uno y hacer, finalmente, una lista con algunas de las opciones.

Accede al documento a través del aula virtual o desde la siguiente dirección web: https://developer.mozilla.org/es/docs/Learn/Server-side/Primeros\_pasos/Web\_frameworks

**Reutilización de software (Ian Sommerville)**

Sommerville, I. (2011). *Ingeniería de software* (9.ª ed.). México: Pearson Educación de México.

El capítulo 16 del libro de Ingeniería del Software de Ian Sommerville, en su novena edición, está íntegramente dedicado a tratar el problema de la reutilización del software. Tras una introducción al tema, el autor dedica el resto de los apartados a profundizar en los *frameworks* de aplicación, las líneas de productos de software y la reutilización de productos.



Desarrollo de software basado en reutilización

Polo Usaola, M. (2012). *Desarrollo de software basado en reutilización* [Material docente]. Barcelona: Universitat Oberta de Catalunya.

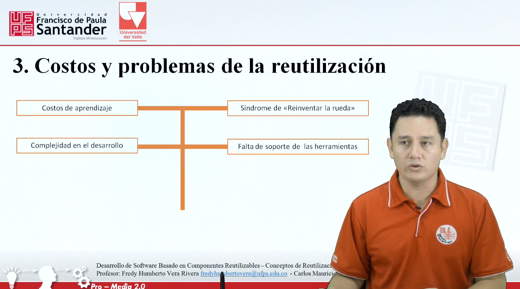
En este documento, que forma parte de la bibliografía del tema, encontrarás ampliamente desarrollados muchos de los conceptos que hemos estudiado.

Accede al documento a través del aula virtual o desde la siguiente dirección web:

<http://hdl.handle.net/10609/63466>

Reutilización de software

En este vídeo el Profesor Fredy Humberto Vera-Rivera realiza una introducción a los conceptos fundamentales de la reutilización del software. Utiliza como referencia fundamental el documento *Desarrollo de software basado en reutilización* (Polo Usaola, 2012).



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

<https://youtu.be/qh4mgcbf5dw>

Test

1. ¿Qué entendemos por reutilización del software?

A. La reutilización de código generado en proyectos anteriores en nuevos desarrollos.

B. La utilización de patrones de diseño y de arquitectura que proporcionan conocimiento acumulado en proyectos anteriores.

C. El empleo de librerías existentes o *frameworks* de desarrollo en la construcción de nuevos sistemas informáticos.

D. Todas las respuestas anteriores son correctas.

1. ¿En qué consiste el enfoque oportunista en el proceso de reutilización de software?

A. En la planificación detallada del proceso aprovechando patrones arquitectónicos.

B. En la mera reutilización de recursos disponibles en nuevos desarrollos cuando esto se muestra necesario.

C. En la subcontratación de ciertas partes del desarrollo que se encarga a terceros.

D. En la utilización de un *framework* de desarrollo de software.

1. ¿Qué ventajas ofrece un buen diseño arquitectónico?

A. Permite reutilizar conocimiento acumulado por expertos en proyectos anteriores.

B. Permite analizar mejor el sistema e identificar oportunidades de reutilización de componentes existentes.

C. Constituyen una potente herramienta de documentación del sistema que ayuda a su comprensión.

D. Todas las respuestas anteriores son correctas.

1. ¿Qué posibles problemas puede plantear el empleo de librerías de enlace dinámico?

A. Problemas de compatibilidad cuando tenemos varios programas que dependen de una misma librería, pero necesitan distintas versiones.

B. Problemas que dificultan la comprensión del código.

C. Las respuestas A y B son correctas.

D. Ninguna de las respuestas anteriores es correcta.

1. ¿Por qué el concepto de herencia de la programación orientada a objetos facilita la reutilización de código?

A. Porque las clases se agrupan en una jerarquía que permite modelar mejor los sistemas.

B. Porque las clases pueden ocultar determinados comportamientos internos que modelan su propio funcionamiento.

C. Porque las clases derivadas heredan comportamientos de los padres que no es necesario volver a implementar.

D. Ninguna de las respuestas anteriores es correcta.

1. ¿Es lo mismo un componente que un servicio?

A. No, porque los componentes no ofrecen servicios al sistema en que se integran.

B. Sí, son lo mismo porque se utilizan del mismo modo en el proceso de desarrollo.

C. No, aunque podemos considerar un servicio como un componente externo del sistema al que ofrece funcionalidades.

D. Sí, porque en ambos casos se emplean las mismas técnicas de reutilización de código.

1. ¿Qué dos tipos de interfaz ofrece un componente software?

A. Una interfaz pública y una interfaz privada.

B. Ofrecen siempre una interfaz «proporciona» y una interfaz «requiere».

C. Una interfaz pública y otra interfaz que ofrece servicios.

D. Una interfaz «proporciona» y, a veces también, una interfaz «requiere».

1. ¿Cuál de las siguientes es una diferencia entre un proceso basado en componentes con reutilización y un proceso de desarrollo más tradicional?

A. El proceso basado en componentes adapta los requisitos en función de los componentes identificados que pueden servir en el sistema actual.

B. El proceso basado en componentes adapta la arquitectura en función de los componentes seleccionados.

C. El proceso basado en componentes emplea siempre metodologías ágiles.

D. Solo las respuestas A y B son correctas.

1. ¿Cuáles son los tres esquemas principales de composición de componentes?

A. Secuencial, jerárquico y aditivo.

B. Secuencial, adaptativo y jerárquico.

C. Integrador, secuencial y aditivo.

D. Aditivo, integrador y analítico.

1. ¿Qué posibles problemas podemos encontrar al desarrollar un nuevo sistema empleando reutilización de software?

A. El sistema puede ser difícil de mantener cuando no tenemos acceso al código fuente reutilizado o conocimientos apropiados.

B. El desarrollo va a resultar siempre más costoso.

C. Puede haber costos de integración al requerirse la adaptación de algunos componentes reutilizados.

D. Las respuestas A y C son correctas.