Universidad Pontificia de Salamanca (Campus Madrid)

Unidad Didáctica 1

Programación de Aplicaciones. Introducción al diseño basado en componentes

UPSAM 2010-2011

Índice de contenido

1. Introducción

Durante el diseño de la **arquitectura software de un sistema** se define un conjunto elementos funcionales clave o esenciales y sus respectivas interrelaciones. El concepto de Arquitectura software es cada vez más importante en el desarrollo de software como campo de estudio, y aunque no existe una definición ampliamente aceptada del término podemos considerar la siguiente [IEE00].

Def. Arquitectura software: *La organización fundamental de un sistema representada a través de sus componentes, sus interrelaciones y los principios que guían su diseño y su evolución.*

Considerando las distintas definiciones se pueden extraer una serie de características importantes:

1. Es una **estructura base fundamental** de diseño.

2. La arquitectura software normalmente está definida en términos de **componentes funcionales, sus interrelaciones y tipos de conectores**. (Nota El concepto de componente en la teoría de arquitectura software es deliberadamente amplio y general para dar cabida a cualquier tipo de elemento clave y significativo).

3. La arquitectura encapsula decisiones y principios de diseño que justifican y guían la construcción del sistema para cumplir con una serie de requisitos tanto funcionales como no funcionales.

En la descripción de una Arquitectura Software toma especial importancia el **concepto de componentes software**. Este concepto en cuanto a significado y caracterización ha ido evolucionando a lo largo del tiempo y existen distintas concepciones de los que es un componente software. Por ejemplo actualmente existen plataformas de desarrollo (Java EE y .NET) que proporcionan una infraestructura de componentes software prediseñados (pre-built components) preparados para ser utilizados en diferentes dominios que en los comienzos no existían y en algunos casos estos componentes constituyen un estándar como por ejemplo los componentes de la plataforma java EE o CORBA.

La concepción de **componente** puede variar bastante pudiendo ser una librería, una clase, un archivo o un servicio web (ej. En una arquitectura orientada a servicios), luego **su interpretación y caracterización dependerá del contexto de interpretación de la descripción de la arquitectura**. Así en un modelo de despliegue tenemos componentes tales como archivos de código fuente (*. C) que serían componentes o en un modelo de realización de un caso de uso tendríamos componentes tales como objetos instancias de una clase.

Una cuestión importante en el diseño de muchos sistemas es la determinar si su diseño y su arquitectura es adecuada para cumplir con una serie de requisitos funcionales y no funcionales requeridos. El diseño a nivel de componentes en este caso representa el software

a un nivel de abstracción tal, que **permite revisar más fácilmente propiedades de diseño de forma que sea posible su evaluación en etapas tempranas de de desarrollo.**

2. ¿Qué es un componente software?

Como hemos comentado **existen varias definiciones del término componente software**. La forma de concebir un componente, muchas veces depende del paradigma de programación empleado (paradigma funcional, imperativo, orientado a objetos) .Además un componente software tiene distintas implementaciones o materializaciones (clases, modulo, funciones, etc) de ahí la dificultad de ofrecer una definición precisa.

En la siguiente figura (Figura 1) se muestra distintos tipos de componentes (hardware y software) formando parte del diseño de distintas arquitecturas.

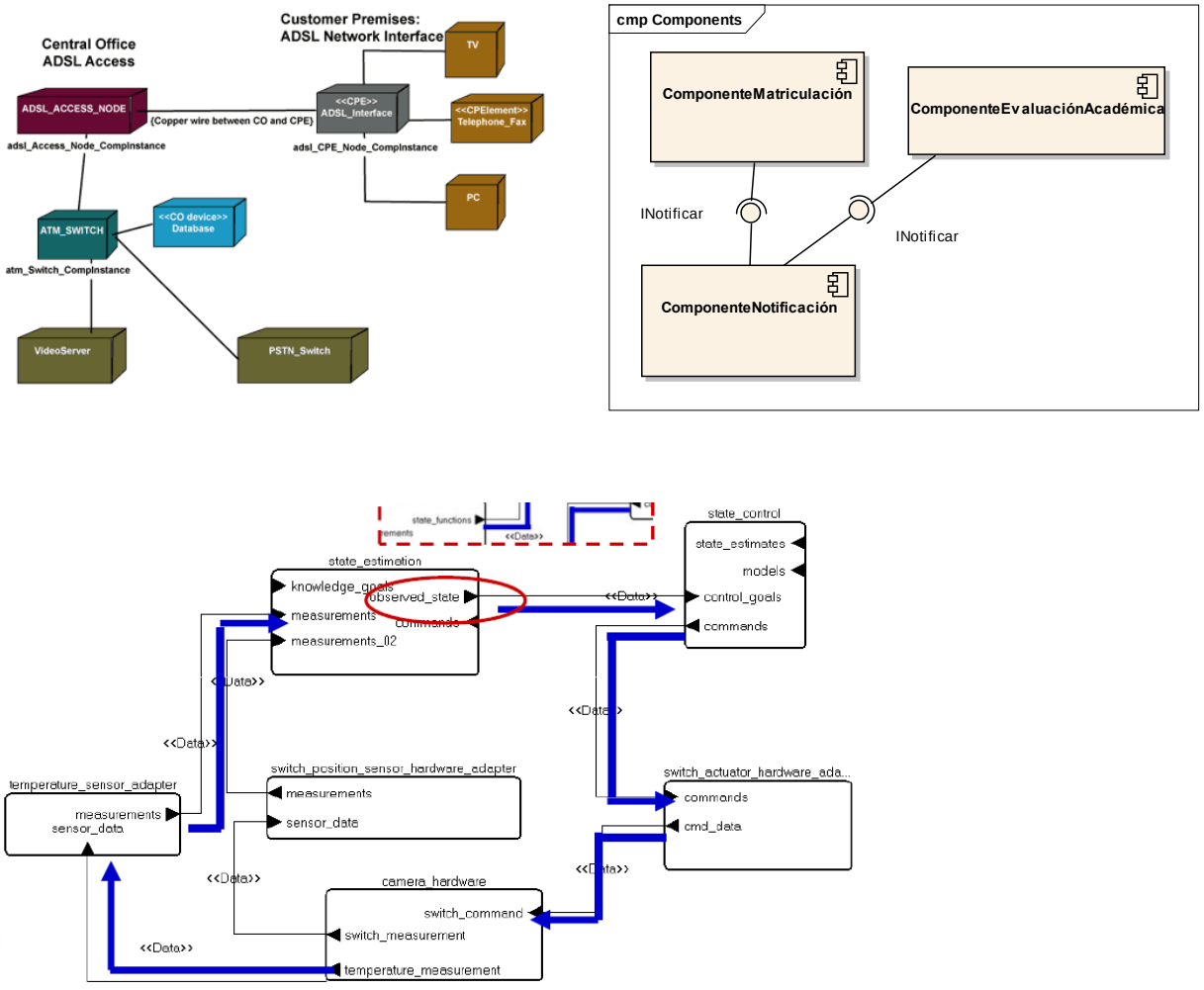


Figura 1 Descripción de sistemas a partir de sus componentes

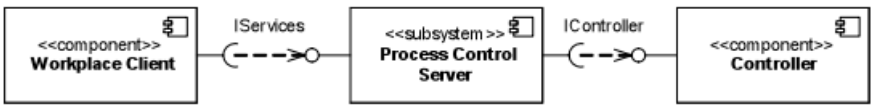


Figure 3: High-level static structure Process Control System.

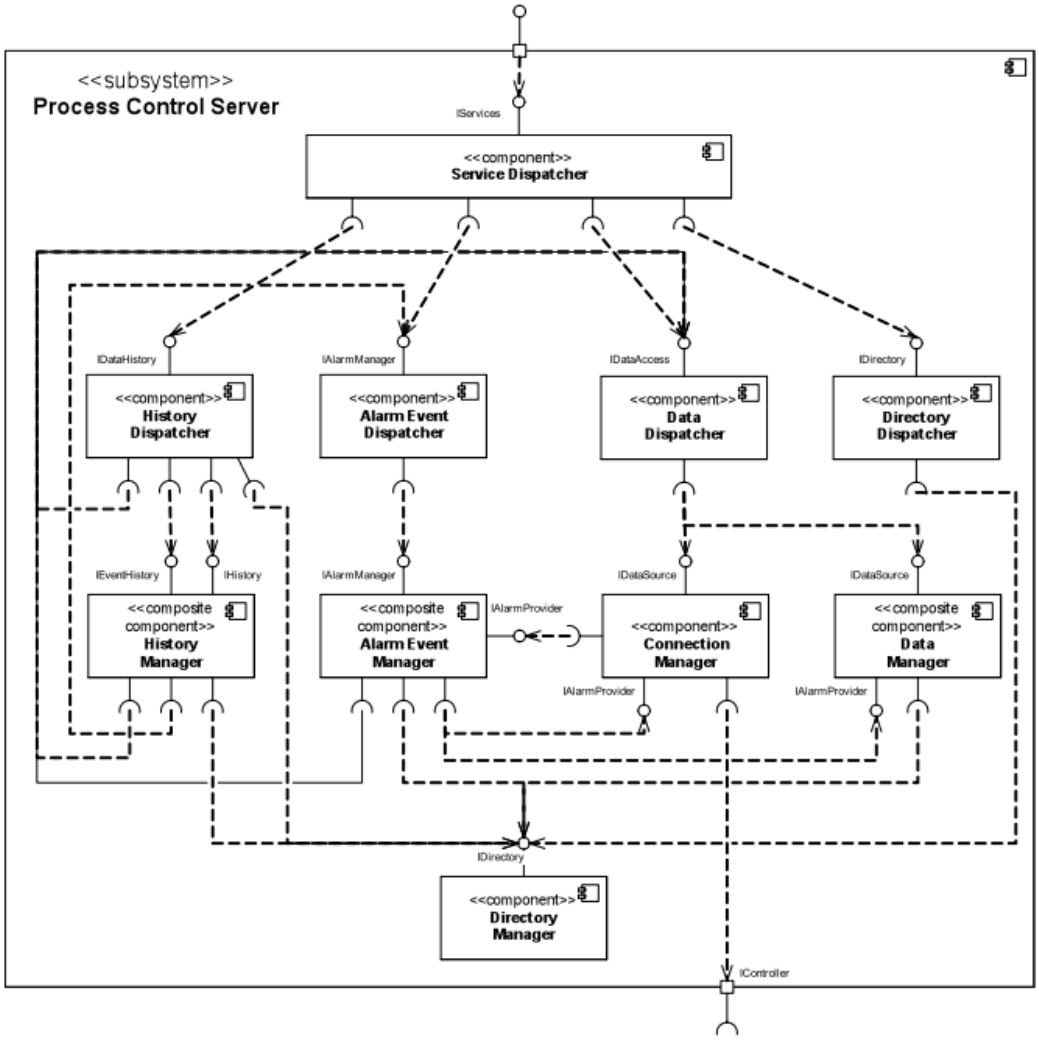


Figure 4: Inner Structure of the subsystem Process Control Server

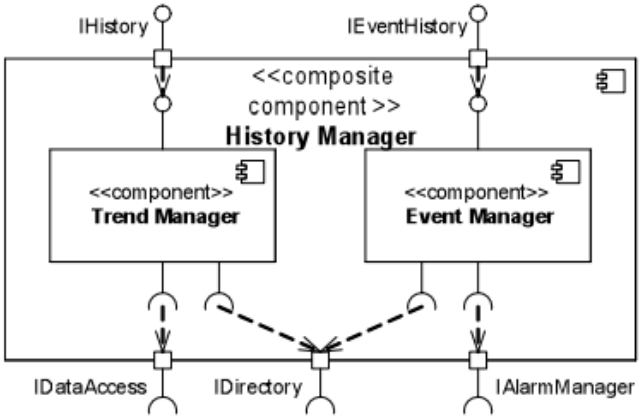


Figure 6: Inner structure of the composite component History Manager

En *"The UML Reference Manual "* **J.Rumbaugh** ofrece la siguiente definición del término componente software:

Un componente software es una parte reemplazable de un sistema que encapsula una implementación y que proporciona la realización de un conjunto de interfaces.

En (Szyperski, 2002) encontramos las siguiente definición

Un componente software es una unidad de composición independiente, con interfaces y dependencias explícitamente definidas y que puede ser desplegado de forma independiente.

Aquí nos centraremos en las ideas que subyacen bajo estas definiciones y su enfoque dentro del paradigma de orientación a objetos. De hecho la idea de desarrollo de componentes esta muy ligada a la programación orientada a objetos.

En primer lugar un componente software es una **pieza reemplazable**. Esta cualidad es muy importante ya que indica que el componente ha sido diseñado con un acoplamiento mínimo. Por otro lado un componente software tiene un **objetivo claro de composición**, esto permite la construcción de forma flexible y modular, en el sentido de que determinadas partes del sistema podrían ser reemplazadas, mejoradas y cambiadas de forma independiente al resto.

En segundo lugar un componente **define e implementa una interfaz**. La interfaz constituye el único medio posible de interacción con el componente (visión de caja negra) .La realización o implementación de una interfaz puede interpretarse por medio de la implementación de un servicio **explícitamente definido** (veremos este cuestión más adelante). Desde el punto de vista orientado a objetos podría concebirse un **componente en diseño como un conjunto de clases que colaboran internamente para implementar algún servicio definido a través de una interfaz.**

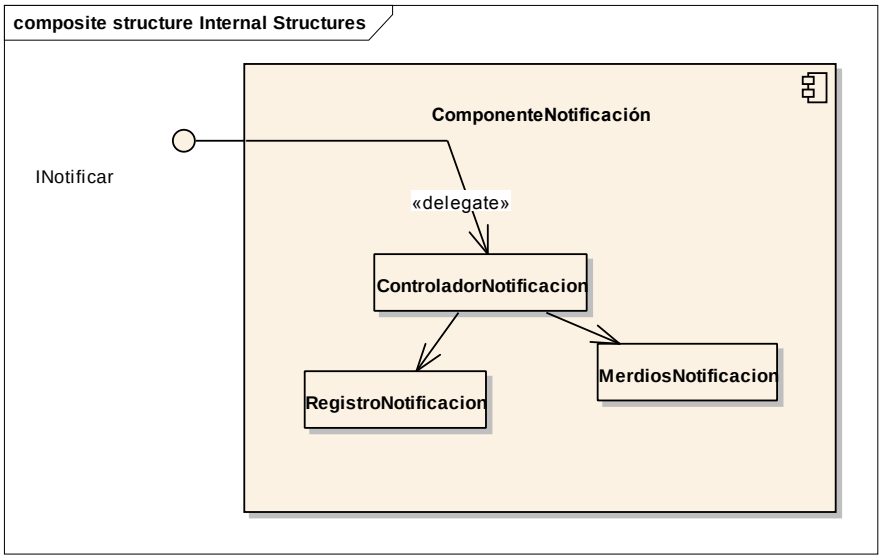


Ilustración 1 Detalle interno de un componente software

Los **componentes software forman parte activa de la arquitectura software** de cualquier sistema, en el sentido de que pueden ofrecer una visión de alto nivel de una parte clave del sistema.

Actualmente existen dos tipos de componentes, aquellos hechos a medida para algún propósito especial y los componentes software estándar (Ej componentes EJB de la plataforma Java EE).

Para terminar, a modo de resumen podemos enumerar las **características** más relevantes de un **componente software** como sigue:

- Diseñado para ser una **unidad modular de composición** en una arquitectura
- **Funcionalidad concreta y bien definida** (diseño altamente cohesivo)
- **Interfaces de acceso explicitas**.
- **Propiedades especificas** (en algunos casos certificadas)
- **Dependencias explicitas**
- **Unidad de diseño independientes**
- **Unidad de despliegue independientes**

3. Componentes. Visión Orientada a objetos.

En el contexto de la ingeniería de software orientada a objetos y más concretamente en el contexto de la arquitectura software de un sistema orientado a objetos, **los propios objetos pueden ser concebidos como componentes software** (de hecho existe un estilo arquitectónico que define precisamente esta forma de construir y concebir sistemas) de una arquitectura. De hecho el paradigma de orientación a objetos es considerado un **estilo de arquitectónico**. No obstante un componente software por su naturaleza podría estar formado por distintas clases de objetos en su estructura interna.

Dentro del diseño de un componente **primará la cohesión funcional** para llevar a cabo una funcionalidad específica de alto nivel. Como parte de este diseño deben definirse o considerarse las interfaces concretas que permiten que los módulos externos se comuniquen con el componente de forma clara y concisa.

El diseño de un componente comenzará con una serie de requisitos funcionales que darán lugar a un conjunto de **clases de análisis que en su elaboración darán lugar a una o varias clases de de diseño que posteriormente se convertirán en clase de implementación**

Para ilustrar todo lo anterior considere el siguiente ejemplo:

*Considere un **componente de impresión de trabajos avanzado**. Donde durante la fase de análisis se obtuvo una clase denominada **ControladorImpresion**. En la parte superior de la **Figura 2** aparecen los atributos y métodos de esta clase. En el diseño de la arquitectura se definió **ControladorImpresion** como un componente. Su representación en forma de componente UML es la que se muestra en la parte derecha. Observe que **ControladorImpresion** tiene dos interfaces **CalcularCosteTrabajo** que proporciona el coste de imprimir un trabajo de impresión, e **IniciarTrabajo**, que pasa el trabajo de impresión a través de las instalaciones de producción.*

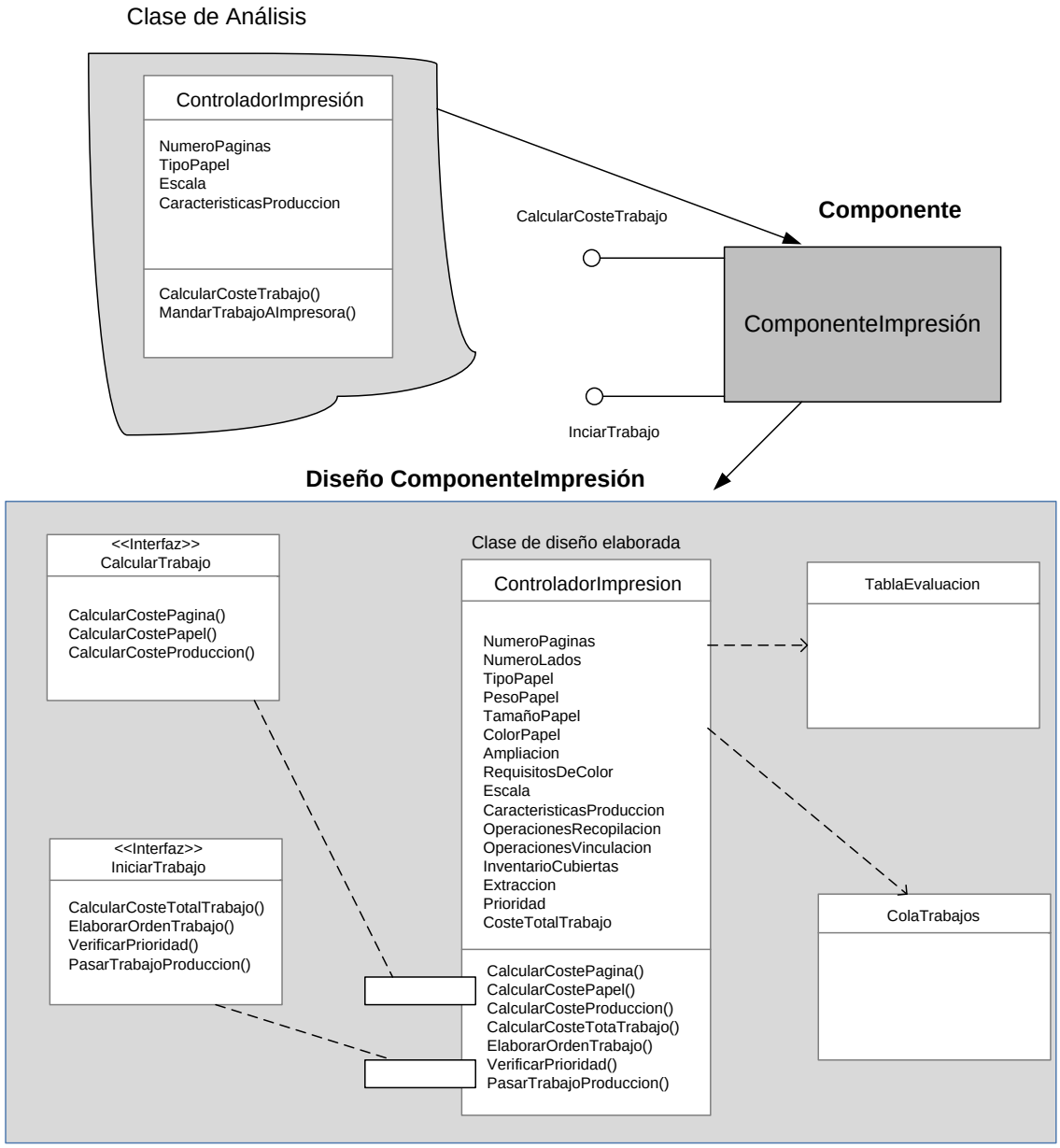


Figura 2 Diseño de componentes

El diseño de componentes software podría comenzar en este punto, una vez identificadas las necesidades y las clases de análisis. Los detalles del componente **ControladorImpresion** deben detallarse para guiar su implementación. La clase de análisis **ControladorImpresion** define los atributos y operaciones con un alto nivel de abstracción, así como el componente que la va a representar. En la parte inferior de la **Figura 2**, se puede ver la clase de diseño ya elaborada **ControladorImpresion,** la cual contiene información más específica y detallada de atributos y operaciones. Las **interfaces CalcularTrabajo e IniciarTrabajo definen de forma abstracta** un servicio que el componente implementará (o realizará) mediante diversos objetos instancia de diferentes clases de control. Así por ejemplo la operación **CalcularCostePorPagina() que es parte de la interfaz CalcularTrabajo** para ser implementada necesita una clase denominada **TablaEvaluación que tiene información sobre los precios.** Otro ejemplo sería

VerificarPrioridad() que conjuntamente con la clase **ColaTrabajos** verifica la prioridad que tiene el Trabajo en la cola de espera.

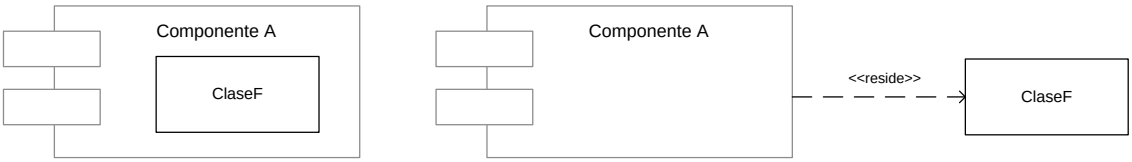
De este diseño se pueden extraer las siguientes características. La **clase "cliente"** de este componente **no será dependiente del diseño interno el componente**. Esto implica que el acoplamiento se reduce entre una la clase "cliente" y el componente. En segundo lugar puesto que las clases clientes de este componente no estarán acopladas al diseño interno del componente, el componente puede rediseñarse o cambiar sin afectar a las clases que usan su interfaz.

Para describir la estructura interna del componente se pueden usar un modelo estructural de clases (UML) y para **describir el comportamiento del componente** en análisis **se podrían usar diagramas de estados, diagramas de actividades o diagramas de colaboración y secuencia**.

Como vemos un **componente software puede considerarse como un sistema a pequeña escala**. De hecho **un subsistema puede ser implementado mediante un componente**.

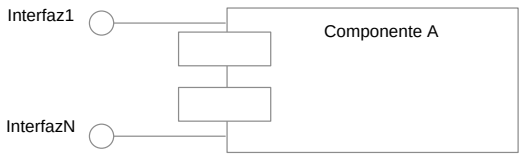
4. Representación UML de componentes

Se puede mostrar la asignación de cada clase al componente como un confinamiento (Figura a), o como una dependencia expresada con el estereotipo **<<reside>>** entre el componente y la clase (Figura b). Los dos estilos son semánticamente equivalentes. A continuación se muestran dos ejemplos:

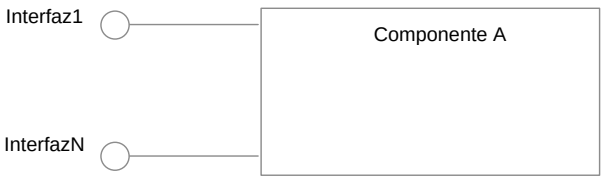


a) Representación con clases confinadas b) Representación usando dependencia

Para indicar las interfaces que implementa el componente se utiliza la siguiente notación UML 1:



En UML 2.0 la notación de un componente es la siguiente



Cada **interfaz** representa un conjunto de **operaciones o servicios abstractos** que son **implementadas por el componente**. Este tipo de representación ofrece una **visión de caja negra** centrada en las propiedades externas (interfaces) y no en sus detalles de diseño e implementación internos.

5. El concepto de interfaz

Según la definición de la DRAE el término interfaz se define como

---

*Conexión física y funcional entre dos aparatos o **sistemas** independientes.*

Según el diccionario Webster:

---

the place at which independent and often unrelated systems meet and act on or communicate with each other<the man-machine interface>

De manera general podemos ver una interfaz de un componente **como un medio** de interacción.

En otro contexto por ejemplo la **interfaz gráfica de usuario** actuaría de **medio** con que el usuario puede comunicarse con un ordenador. De forma general **existen distintos tipos de interfaces**: interfaces hardware, interfaces software, interfaces gráficas de usuario, etc. A su vez dentro de las interfaces software existen distintos tipos de interfaces.

En este tema trabajaremos con **interfaces como elementos de los lenguajes de programación orientadas a objetos.** En los lenguajes orientados a objetos el concepto de interfaz tiene distintas interpretaciones. La interfaz de una clase, la constituyen el conjunto de métodos públicos que expone la clase para ser utilizada. Sin embargo en algunos lenguajes como Java y C# el término **interfaz** define un **elemento propio lenguaje** que **sirve para definir un comportamiento abstracto u operaciones abstractas**.

5.1. Interfaces y componentes

De forma simple la interfaz de un componente **es el medio** por el cual se puede producir una interacción con el componente. De manera más formal y desde una visión orienta a objetos.

Una interfaz es un conjunto de operaciones abstractas que serán implementadas por alguna clase de implementación.

La interfaz del componente es la especificación de un conjunto de operaciones abstractas, que son implementadas por el componente internamente. **El objetivo** de la interfaz de un componente es el de **separar la *especificación* de un comportamiento** o funcionalidad **respecto de su forma de implementación concreta**.

En el diseño a nivel de componentes la **interfaz forma un "pseudocontrato" entre el módulo cliente y el componente** que además de ser cumplido funcionalmente puede implicar restricciones y condiciones de funcionamiento.

El pseudocontrato anterior **puede ser descrito o establecido en forma de pre-condiciones y post-condicones** para cada operación de la interfaz. Así tendremos que:

- El cliente tendrá que garantizar las precondicones antes de poder ejecutar la operación de la interfaz
- El proveedor de la implementación (el componente) debe confiar en que se cumplen las precondicones antes de llevar a cabo la operación.
- El proveedor de la implementación tiene que establecer las post conciones antes de devolver control.
- El cliente puede confiar en que el componente cumple con las postcondicones después de ejecutar las operaciones

En definitiva y más precisamente la implementación de un contrato debería llevar asociadas las siguientes características

- Características funcionales
- Requisitos no funcionales asociados (memoria usada, tiempo de respuesta, seguridad etc)
- Precondiones y postcondiones

Las interfaces tienen un gran impacto en la forma de diseñar colaboraciones entre clases de objetos. Hasta ahora se han **diseñado colaboraciones** directas entre clases objetos. Sin embargo en muchos caos **es más flexible diseñar interacciones mediante interfaces** propias del lenguaje y que estas interfaces puedan estar implementadas por distintas clases. Por ejemplo supongamos un sencillo juego donde diversos elementos deben ser posicionados en distintas posiciones a los largo del desarrollo del juego. Estos elementos móviles podrían implementar todos la misma interfaz ***cambiarPosicion(inc x, inc y)*** que permite cambiar el estado de todos los elementos susceptibles de ser movidos espacialmente en función de algún tipo de algoritmo de IA.

Ejemplos más concretos de interfaces los tememos en las distintas plataformas de desarrollo orientadas a objetos como por ejemplo Java. Así en Java cualquier objeto que requiera ser guardado en el sistema de archivos debe implementar obligatoriamente la interfaz **java.io.*serializable.*** Otro buen ejemplo es el conjunto de clases de tipo colección de java , en este caso **hay diez implementaciones de colecciones** con características internas distintas y todas implementan la interfaz **Collection** que define los **métodos abstractos** size(),add() entre otros.

En la plataforma .NET por ejemplo para implementar un manejador de solicitudes Http en el entorno ASP.Net un componente debe implementar la interfaz **IHttphandler obligatoriamente.**

Las **interfaces** se convierten en elementos de la arquitectura muy importantes en diseño, debido a que proporcionan **una especie de conector abstracto que permite a los desarrolladores conectar sus subsistemas sin depender de clases de implementación concreta o específica**. Además de esta funcionalidad, las interfaces se usan para hacer que un conjunto de **clases implementen un mismo comportamiento abstracto**, de forma que puedan ser tratadas de manera homogénea (**polimorfirsmo**) con distinto comportamiento interno.

Es importante recordar en este punto que **una interfaz define sólo prototipos de operaciones**, **semántica, estereotipos y restricciones**. **No especifica, ni implica ninguna implementación**.

En general cada operación de una interfaz (OO) ***debe contener***:

- **El prototipo de la operación**: nombre, parámetros y tipo de retorno
- **La semántica de cada operación**: Esto puede indicarse como una nota de texto o pseudocódigo; opcionalmente un estereotipo y un conjunto de restricciones

Por otro lado las interfaces ***no pueden tener***

- Atributos (salvo atributos constantes)
- Implementación de las operaciones
- Asociaciones con clases de objetos

Conceptualmente una **interfaz (OO) es una clase abstracta** con la excepción de que no tiene atributos y todas las operaciones que especifica son abstractas (recordamos que una clase abstracta, por ejemplo en c++ puede tener implementación de operaciones). Sin embargo, lo que hace a las interfaces diferentes para indicarlas y considerarlas de forma diferenciada de una clase abstracta es que **tienen la restricción de que no pueden definir ninguna asociación** con otras clases.

Simbólicamente un interfaz se puede representar de diversas formas en UML **(Figura 3)**

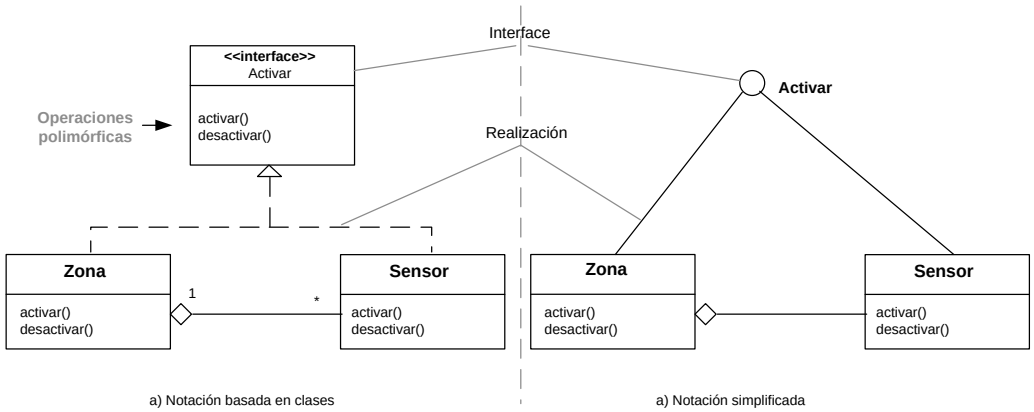


Figura 3 Representación de interfaces y sus clases de implementación

La **asociación de realización/implementación** muestra una especificación de un servicio de forma abstracta y los elementos (clases) que realizan esa especificación (en este caso las clases Zona y Sensor).

Observando la **Figura 3** se muestra como la interfaz **Activar** define un servicio de activación y desactivación abstracto, que la clase **Zona** y **Sensor** implementan de forma concreta. En el ejemplo este sencillo sistema de control trabaja con agrupaciones de sensores en diversas zonas y desde la perspectiva del sistema activar una zona es equivalente a activar todos y cada uno de los sensores que abarca dicha zona de ahí que ambas clases implementen el mismo servicio.

Se debe notar que en la notación basada en clases, la **relación implementación de un interfaz se dibuja con trazo discontinuo**, mientras que en la notación simplificada es con una línea continua. La primera notación es útil cuando se quiere mostrar las operaciones que definen la interfaz, y la notación simplificada, cuando este detalle no tiene relevancia y el diagrama se quieren mostrar de una manera más simple. Aunque ambas formas de expresión son semánticamente equivalentes.

Un componente que implementa una interfaz puede a su vez hacer uso de una interfaz para comunicarse con otro componente. En la siguiente Figura (Figura 4) se muestra la representación de un componente (Garage Door) y su relación con dos interfaces: **IMovementControl** es implementado por el componente y **ISensor** es usado por el componente. En este sentido se puede hablar de **interfaz requerida** (*Isensor*) e interfaz **implementada** (*IMovemenetControl*) o proporcionada.

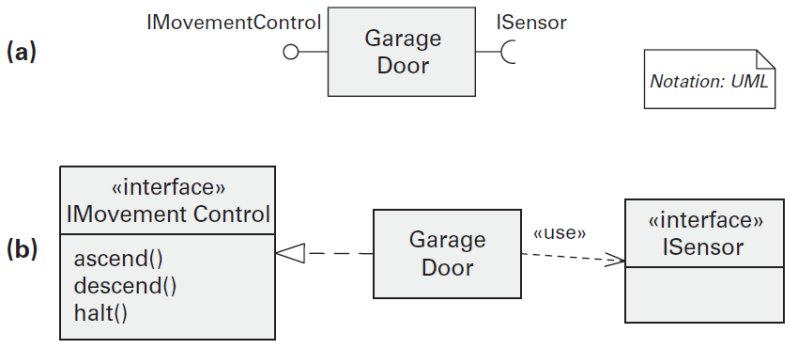


Figura 4 implementación y uso de interfaces por un componente

Las interfaces en algunos lenguajes como C**# y Visual Basic las interfaces llevan el prefijo "I"** delante del nombre de la interfaz. Así en el ejemplo anterior tendríamos la interfaz **"IActivar".**

5.2. Interfaces y diseño de componentes

Las interfaces son un concepto clave en el diseño de componentes CBD (Component –Based Development) como estamos viendo. Si se quiere crear software **flexible** en el sentido de que se puedan intercambiar distintas implementaciones, se deberá hacer uso de interfaces. **Esto es debido a que como hemos visto una interfaz sólo específica un pseudocontrato, permitiendo que cualquier número de implementaciones puedan realizar ese pseudocontrato.**

El uso de interfaces está muy ligado a la especificación de la comunicación entre subsistemas como podemos apreciar en el siguiente diagrama (Figura 5). De hecho podemos implementar un subsistema como un componente.

En la siguiente figura (Figura 6) podemos definir de forma genérica que el subsistema planificador de rutas usa la interfaz GeoPosicionamiento para interactuar con el subsistema de Localización Geográfica

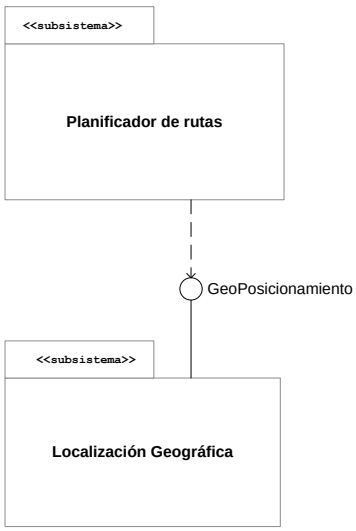


Figura 5 Interfaces y subsistemas

El siguiente ejemplo **(Figura 6)** se muestra un sistema de seguridad de oficinas de una compañía. Cada planta del edificio se considera una zona (clase **Zona**). Cada **Zona** tiene un lector de tarjetas que activa o desactiva identificaciones de tarjetas para cada zona. En este modelo de sistema, cada zona puede verse como una conjunto de elementos que pueden ser activados y desactivados (implementación de la interfaz IActivar). Esto hace que sea más fácil conectar nuevos dispositivos tales como alarmas de incendios u otros dispositivos que puedan ser activados o desactivados.

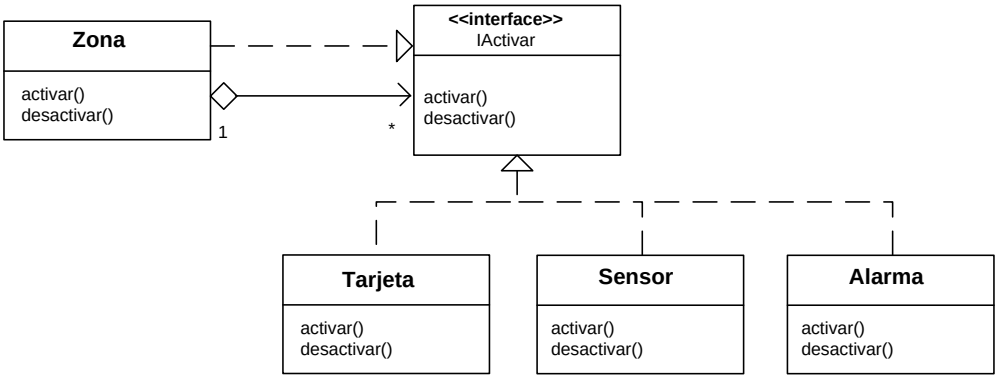
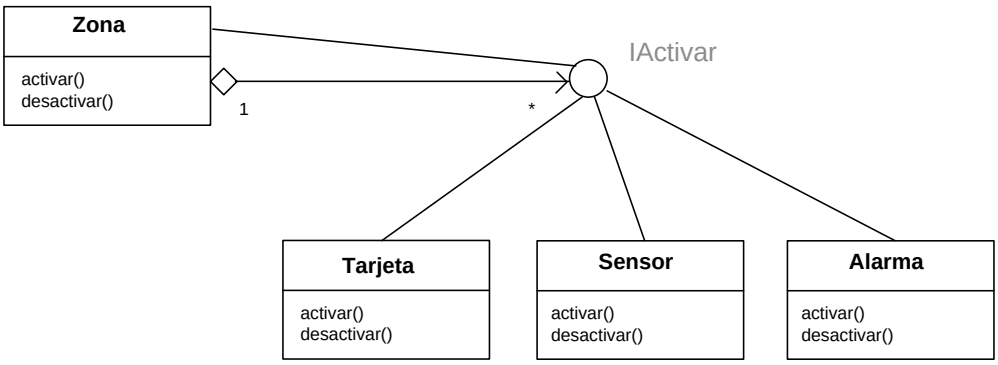


Figura 6 Ejemplo de utilización de interfaces

Usando la notación simplificada tendríamos:



En este ejemplo hay que considerar que la relación entre **Zona** y la interfaz **I*Activar*** es unidireccional. Esto es necesario porque aunque las clases que implementan la interfaz pueden tener asociaciones con otras clases, un interfaz en sí misma no puede contener asociaciones.

5.3. Consideraciones del diseño con interfaces

Cuando se tiene que diseñar un sistema o una parte de un sistema, merece la pena examinar y encontrar las interfaces que pudieran mejorar el diseño. Para realizar esta tarea se podría considerar lo siguiente:

1. **Cuestione el sentido de cada asociación**. Pregúntese lo siguiente **¿Esta asociación debería hacerse con una clase de objetos particular o debería ser una asociación más flexible incluyendo otro tipo de clase de objetos?** Si la respuesta es afirmativa considere incluir una interfaz en la asociación.

2. **Cuestione el sentido de cada intercambio de mensajes**. Pregúntese lo siguiente. ¿Este mensaje debería enviarse a una sola clase de objeto o debería ser más flexible incluyendo más clases de objetos? Si el mensaje debe ser más genérico (Es decir si puede tiene cabida el planteamiento de que un mismo mensaje puede ser enviado a distintas clases de objetos) puede considerar usar interfaces.

3. **Considere factorizar grupos de operaciones que se utilizan en varios casos para que puedan generalizarse**. Por ejemplo si existen varias clases en el sistema que requieren la capacidad de imprimirse hacia algún dispositivo, considere definir un interfaz Imprimir para que sea implementado por cada clase. Esto permite homogeneizar y formalizar un comportamiento determinado

4. Busque **clases que jueguen el mismo Rol** en el sistema. Este Rol podría indicar la definición de una interfaz. Ejemplos (clases que hacen entrada de datos, o salida de datos, etc).

5. **Considere las posibles extensiones del sistema.** Algunas veces con un poco de visión de futuro, se pueden diseñar sistemas que pueden extenderse fácilmente en el futuro. La clave es "en el futuro" en los requisitos, ¿Necesitaré añadir otro tipo de clases al sistema? Si la respuesta es afirmativa, se debe intentar definir una o más interfaces que definan un comportamiento que será implementado por esas nuevas clases y que ya está definido en el diseño del sistema.

5.4. Ventajas y desventajas del uso de interfaces

El diseño muchas veces está restringido a usar una serie de implementaciones específicas. Pero cuando se **diseña con la mirada puesta en interfaces, el diseño se puede basar en ciertos pseudocontratos (visto anteriormente) que pueden ser realizados por diferentes implementaciones**. El **diseño por contrato libera al modelo de diseño** (y en última instancia al sistema) **de dependencias de implementación**, incrementando su flexibilidad y extensibilidad futura. El diseño por contrato incluye no solamente la implementación de una interfaz, sino que también incluye el cumplimiento de precondiciones y postcondiciones e invariantes, cuestiones claves en la fiabilidad del sistema.

El diseño con interfaces permite reducir el número de dependencias entre clases, subsistemas y componentes. Debido a esto, este tipo de diseño ofrece un cierto control sobre el acoplamiento en un modelo. Así un uso adecuado de interfaces ayudan a reducir este acoplamiento y a separar el modelo en subsistemas cohesivos y débilmente acoplados.

Sin embargo existen una serie de **desventajas** en el uso de interfaces. En general cuando se desea que algo sea más flexibles, se consecución e implementación se hace más compleja. Cuando se diseña con interfaces, **se debe buscar una solución de compromiso entre flexibilidad y complejidad**.

6. Principios básicos de diseño de clases

Hay **una serie de principios** básicos que son aplicables a cualquier diseño OO y por ende al diseño de componentes .La motivación subyacente para aplicar estos principios es **crear diseños que más fáciles de extender y modificar**, así como la de **reducir los efectos colaterales** cuando se realicen cambios. Sin embargo **es difícil encontrar una serie de criterios que definan un buen diseño**. , quizás sería más inmediato definir una serie de criterios de un mal diseño como por ejemplo:

1.- Es **difícil realizar cambios** porque cada cambio afecta a muchas otras partes del sistema (R**igidez**).

2.- Cuando se realiza un cambio, **de manera inesperada partes del sistema comienzan a fallar (Fragilidad)**

3.-**Es difícil reutilizar** partes en otras aplicaciones porque no se pueden independizar debido alto acoplamiento (**no reutilización**)

4.- No se dispone de una estructura clara de organización

Existen una serie de principios de diseño que se pueden aplicar en general a cualquier diseño OO. Aquí tomamos los definidos en *Agile Software Development: Principles, Patterns, and Practices (Robert C. Martin, Prentice Hall, 2002)*. Estos principios de diseño son:

- OCP: The Open/Closed Principle
- SRP: The Single Responsibility Principle
- LSP: The Liskov Substitution Principle
- ISP: The Interface Segregation Principle
- DIP: The Dependency Inversion Principle

6.1.1. Principio Abierto-Cerrado (OCP Open-Closed Principle)

"Un módulo debe ser abierto a su extensión pero cerrado a su modificación".

Este principio fue definido por **Bertrand Meyer** y significa que los componentes deberían estar abiertos para que se puedan extender o adaptar pero por otra parte deberían permanecer cerrados a las modificaciones internas ante estas adaptaciones o extensiones.

Este enunciado parece una contradicción, pero representa una de las características más importantes de un buen diseño de componentes. Dicho en otras palabras significa que debe especificarse el componente de forma tal que permita extenderlo (dentro del dominio funcional que al que está dirigido) sin necesidad de hacerle modificaciones internas (a nivel de lógica).L**a abstracción será clave** dentro de este principio

Por ejemplo suponga un sencillo módulo denominado **SupervisorInstalacion** que se encarga de comprobar el estado de cada tipo de sensor instalado. Supongamos que el número y tipo de sensores puede variar. Podríamos considerar en primera instancia una lógica de control para este componente como sigue.

```
Si (sensor.tipo()=tipo1) entonces

//instrucciones para sensor tipo1

Si (sensor.tipo()=tipo2) entonces

//instrucciones para sensor tipo2

...

Si (sensor.tipo()=tipoN) entonces

//instrucciones para sensor tipoN
```

Como se puede observar **el código de control es dependiente del tipo** concreto de sensor. La introducción en la instalación de un nuevo tipo de sensor implica cambiar la implementación de la clase **SupervisorInstalacion. Esto sería una violación del principio abierto-cerrado en el componente.**

En la **Figura 9** se ilustra una forma de cumplir con este principio de diseño para la clase **SupervisorInstalacion**. La **interfaz Sensor presenta una visión homogénea** y consistente de cada uno de los sensores a nivel de funcionalidad. Si se agregase un nuevo tipo de sensor, no se requeriría hacer ningún cambio para la clase **Detector.**

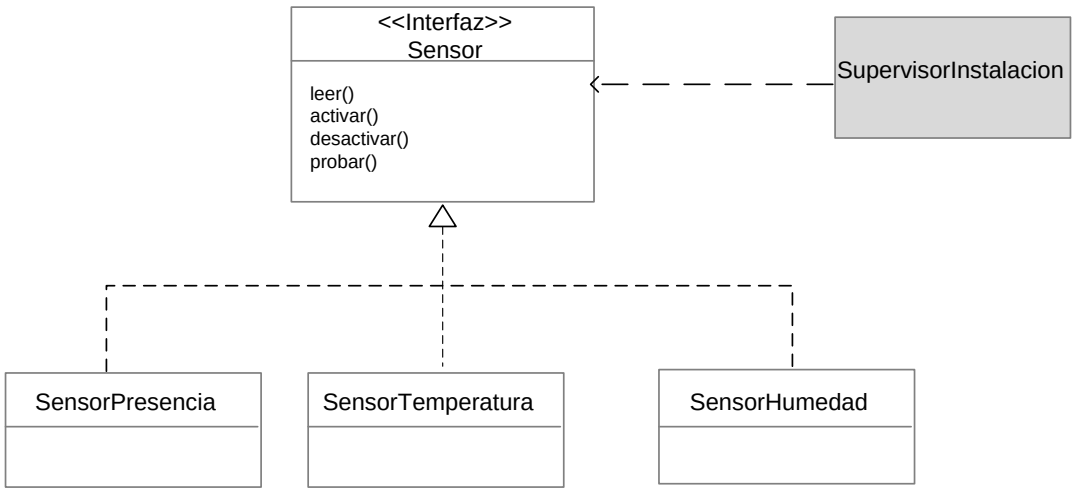


Figura 7 Principio abierto cerrado a nivel de diseño de componentes

Como vemos existen **dos conceptos claves** para implementar este principio en POO como son **la abstracción (interfaz) y el polimorfismo**.

Conclusiones

La consideración del principio OCP en el diseño, lleva a la **creación de módulos extensibles y flexibles**. Esto significa que con un poco de previsión se pueden añadir nuevas características a un diseño, sin cambiar el código existente.

Se debe considerar por otra parte que seguir este principio de diseño no significa que el sistema deba ser 100% cerrado (esto sería muy complejo de diseñar).

Por ejemplo imaginemos un función **dibujar(Figura f) del ejemplo dado en clase**, que debe dibujar primero las figuras de tipo cuadrado y después los círculos. Esta función no podría ser cerrada ante un cambio como este. En general no importa como de cerrado es un módulo, **siempre existirá algún tipo de cambio contra el cual no se puede ser cerrado.**

Puesto que el cierre total de un módulo no puede considerarse a veces viable, esta decisión debe ser algo estratégico. El diseñador debe elegir qué tipo de cambios serán admisibles en el diseño. Esto implica práctica y experiencia

6.1.2.Principio de Responsabilidad única (SRP Single Responsability Principle)

Este principio también es conocido como **principio de alta cohesión** y como el principio abierto cerrado es un principio fundamental de diseño.

Una clase debería ser cohesiva, esto es, **debería tener un único propósito funcional y todos los métodos deberían estar diseñados para alcanzar ese propósito**. O en palabras de Robert C Martin. **" No debería haber más de una razón para que una clase cambiase ..".**

Cuando una clase tiene más de una responsabilidad hace que la clase sea más difícil de entender, más complicada de modificar, y más difícil de reutilizar (rigidez y complejidad innecesaria).

Por ejemplo considere una clase denominada **ConexionBD** que permite distintas funcionalidades con una base de datos relacional. El diseño de esta clase se muestra en la siguiente figura **(Figura 10).**

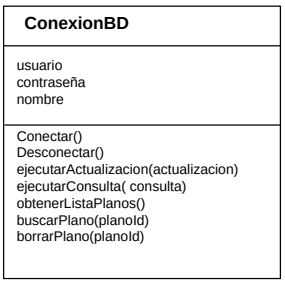


Figura 8 Diseño que no sigue el principio SR

Esta clase no se considera cohesiva a nivel de diseño , ya que por un lado tiene implementadas responsabilidades funcionales que tienen que ver con la conexión y por otro lado

responsabilidades de realizar tareas de gestión sobre planos. La solución como muestra la siguiente figura **(Figura 11)** sería hacer que la clase fuera cohesiva y realizase una sola tarea funcional. La responsabilidad de gestión de planos en este caso podría delegarse a otra clase distinta.

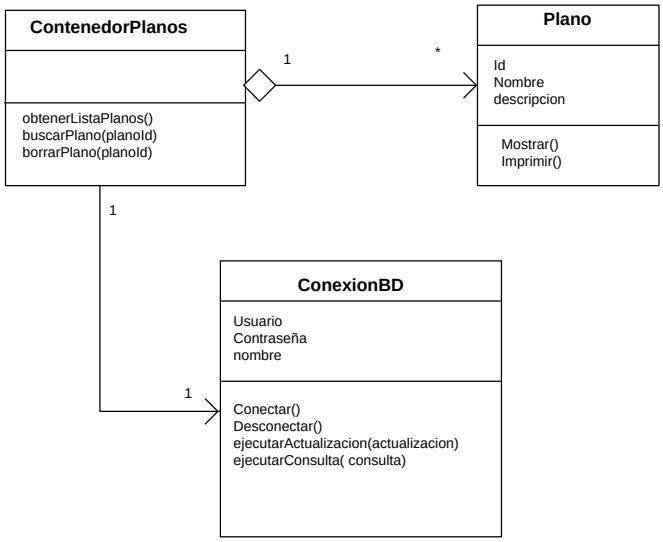


Figura 9 Diseño de única responsabilidad

El **reto** en el diseño de este principio es **proporcionar una división de responsabilidades adecuada** a nivel clase.

Por último, acerca del principio SRP, se debería considerar que si no se puede dividir las responsabilidades en diferentes clases, al menos, se debería considerar separarlas mediante interfaces diferentes. Este principio es conocido como **Principio de Segregación de interfaces** (que veremos más adelante).

6.1.3.Principio de sustitución de Liskov (LSP Liskov Substitution Principle)

Este principio de diseño fue acuñado por **Barbara Liskov** en un trabajo sobre teoría de tipos y abstracción de datos. También se deriva de ideas sobre el concepto de diseño por contrato (DBC) de Bertrand Meyer.

En 1987**, Barbara Liskov** presentó en una conferencia la siguiente definición del principio LSP:

"S es un subtipo de T si por cada objeto o1 del tipo S existe un objeto o2 del tipo T tal que para todo programa P definido en términos de T, el comportamiento de P permanece invariable cuando o1 es sustituido por o2".

En un lenguaje más coloquial y aplicado al concepto de clase significa que: **"cualquier subclase debería poder usarse de forma consistente allí donde se use su clase base"**. Lo que implica que **las subclases deben preservar un comportamiento y una serie de restricciones definido por la clase base a la que pertenecen**.

Este principio de diseño sugiere que un componente que use una clase base debe funcionar también con una subclase derivada. También implica que cualquier clase derivada de una **clase base debe respetar cualquier contrato implícito** entre la clase base y los componentes que la usan. En un contexto de análisis, ese "contrato" podría ser una precondición que debe ser verdadera antes de que el componente use la clase base y una pos condición que debe ser verdadera después de ello .

En la literatura clásica se muestra el siguiente ejemplo que ilustra la violación del principio LSP. En este ejemplo se muestra lo que se considera una incorrecta implementación de la herencia.

Considere una aplicación que trabaja con figuras geométricas rectangulares. El sistema se amplia para que trabaje con cuadrados también. Claramente un cuadrado es un rectángulo. Puesto que existe la relación lógica "es un tipo de" mediante la herencia, es lógico modelar un Cuadrado como una clase derivada de Rectángulo (Figura 10).

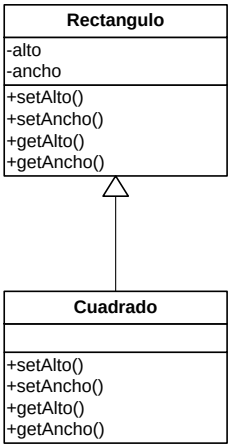


Figura 10 Un Cuadrado es un Rectángulo

La justificación de la jerarquía es que un cuadrado es un rectángulo, pero veremos más adelante que este tipo de planteamientos tiene algunos matices sutiles. Generalmente **muchos problemas de diseño no se identifican hasta que se realiza una implementación de una aplicación.**

La implementación de estas clases en Java es la siguiente:

```
public class Rectangulo {

    protected int ancho;
    protected int alto;

    public Rectangulo() {
    }

    public int getAlto() {
        return alto;
    }

    public void setAlto(int alto) {
        this.alto = alto;
    }

    public int getAncho() {
        return ancho;
    }

    public void setAncho(int ancho) {
        this.ancho = ancho;
    }

}
```

```
public class Cuadrado extends Rectangulo {

    public Cuadrado() {
    }

    @Override
     public int getAlto() {
        return super.alto;
    }
```

```
    @Override

    public void establecerAlto(int alto) {
        super.alto = alto;
        super.ancho=alto;
    }
    public void establacerAncho(int ancho) {
        super.ancho = ancho;
        super.alto=ancho;
    }


    @Override
    public int Ancho() {
        return super.ancho;
    }



}
```

La primera pista de este diseño incorrecto, viene del hecho de que la clase **Cuadrado** no necesita dos variables para representar sus características como la clase **Rectángulo**. Aún así se puede hacer alguna salvedad a este respecto. Otra de las cuestiones es que Cuadrado hereda las funciones **setAlto() y setAncho()** aunque estas funciones no llegan a ser apropiadas para la clase Cuadrado. Esto sería una pista suficiente para indicar que hay algún problema de diseño con la jerarquía diseñada. Sin embargo existe una forma de solventar el problema se puede sobrescribir los métodos **setAlto() y setAncho()** como sigue:

```
@Override
    public void setAlto(int alto) {
        superarte = alto;
        super.ancho=alto;
    }

    @Override
    public int getAncho() {
        return super.ancho;
    }
```

Ahora cuando alguien cambie el ancho de un Cuadrado, también se cambiará el alto y viceversa. Así los invariantes de un objeto Cuadrado quedan intactos. El objeto Cuadrado continua siendo correcto matemáticamente.

```
Cuadrado c= new Cuadrado();
c.setAlto(4) ; //Se establece en ancho al mismo valor
c.setAncho(3)// Se establece el alto también
```

Problema

En este punto tenemos dos clases Cuadrado y Rectángulo que parece que funcionan. No importa los que se haga con el objeto Cuadrado, continúa siendo consistente matemáticamente hablando. Incluso se podría pasar un objeto Cuadrado a una función que acepte un puntero a un Rectángulo, y el objeto rectángulo actuaría como un Cuadrado y continuaría siendo consistente.

De esta forma se podría concluir que el modelo es consistente en sí mismo y correcto. Sin embargo esta conclusión podría verse contradicha. **Un modelo que es consistente en sí mismo puede no serlo con todos sus usos**. Para ello considere la siguiente la siguiente función:

```
public static void g (Rectangulo r){

      r.setAlto(5);
      r.setAncho(4);
      assert ( r.getAlto()*r.getAncho()==20);
    }
```

Esta función trabaja bien para un **Rectángulo**, pero **dará error** si se pasa un **Cuadrado**. El desarrollador asume al crear esta función que el tipo de parámetro será un Rectángulo siempre. Claramente el programador de la función g hizo una asunción razonable. Pero las asunciones pueden fallar como en el ejemplo y si a esta función se usa para un Cuadrado no podrá operar apropiadamente. Este tipo de funciones suponen una violación del principio LSP. Además el principio abierto cerrado será violado también puesto que la adición de un nuevo tipo de figura cambiaría la lógica de esta función.

Esto nos lleva a una importante conclusión, **un modelo analizado aisladamente no significa que sea válido**. La validez de un modelo sólo puede ser expresada en términos de sus clientes (es decir las clases que van a usar el modelo).

Pero ¿Cuál ha sido el problema de este modelo? Después de todo un **Cuadrado** es un **Rectángulo**. Según lo visto la respuesta es no, porque el comportamiento de un Cuadrado no es consistente con el comportamiento de un Rectángulo**. Lo que implica que una subclase no sólo debe ser consistente con los atributos heredados, sino también con el comportamiento que hereda y que debe desarrollar.**

Desafortunadamente, las violaciones de este principio son difíciles de detectar hasta que ya es demasiado tarde. Si el diseño empleado es importante, el coste de reparar esta violación puede ser demasiado grande. La solución pasaría por poner una estructura if/else en el cliente para comprobar si es verdaderamente una elipse y no un círculo , pero esto a su vez causaría una violación del principio abierto-cerrado

6.1.4.Principio de segregación de interfaz (ISP Interface Segregation Principle)

"Muchas interfaces especificas es mejor que una única interfaz de propósito general"

Mientras el principio de diseño SRP está dirigido a alcanzar una gran cohesión a nivel de clases, el principio ISP se ocupa de la **cohesión a nivel de interfaces**. El objetivo fundamental de este principio es el de crear interfaces especificas para cada cliente de la clase o componente, para que estos clientes no dependan de funcionalidad que no necesitan.

Este principio sugiere que debe crearse una interfaz especializada que atienda a cada categoría de cliente. En la interfaz de un cliente particular sólo deberán abstraerse las operaciones necesarias para ese cliente en particular. Cada interfaz por tanto debería tratar con un aspecto

específico de comportamiento. Por ejemplo dado la interfaz **ImprimirTrabajo (Figura 8)** con el siguiente conjunto de operaciones:

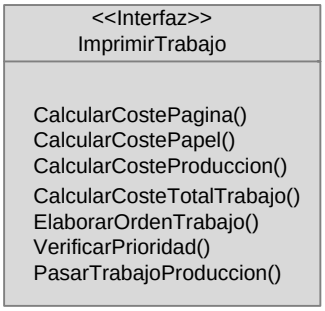


Figura 11 Interfaz no cohesiva

Como se puede observa se está vulnerando el principio de segregación de interfaz. Puesto que el cálculo del coste del papel es un comportamiento y una funcionalidad diferente a la de enviar el trabajo de impresión a la impresora.

La solución a este problema de diseño sería segregar las operaciones de la interfaz en varias interfaces (CalcularTrabajo e iniciarTrabajo), como se muestra a continuación.

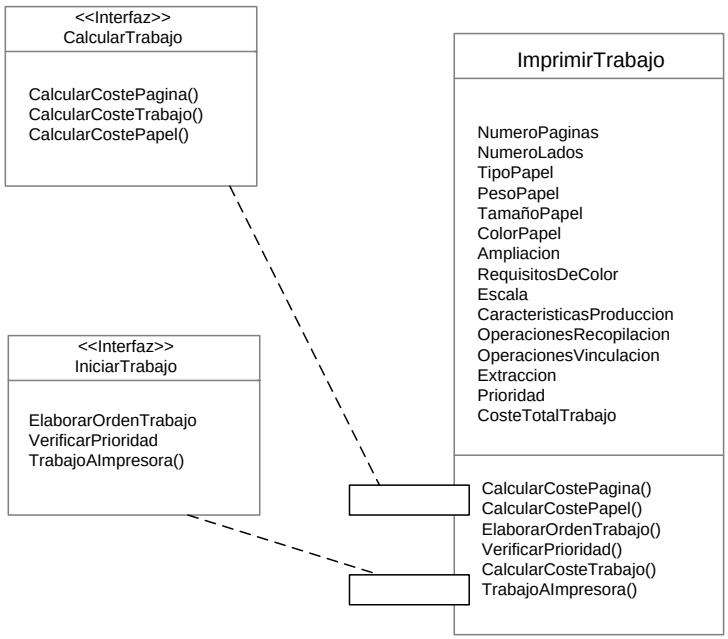


Figura 12 Segregación de interfaces

En este caso vemos que la interfaz **ImprimirTrabajo** se pueden segregar en dos interfaces, Una interfaz que agrupa las operaciones relacionadas con el coste del trabajo de impresión y otra con las operaciones relacionadas con el envío de la trabajo de impresión a la impresora con una configuración determinada. Bajo el principio de segregación de interfaz como vemos se obtiene dos interfaces cohesivas diferentes **CalcularTrabajo** e **InciarTrabajo.**

El principio ISP se puede utilizar como una solución donde el diseño de una clase viola el principio SRP por alguna razón.

6.1.5.Principio de inversión de dependencia (Dependency Inversion Principle)

Uno de los principios más simples y fundamentales del diseño de objetos es el principio de inversión de dependencia (No se debe confundir este principio con el de inyección de dependencia (DI Dependency Injection o Inversión Of Control).

Enunciado *"Los módulos de un determinado nivel de abstracción no deberían depender de detalles de bajo nivel, sólo deben depender de abstracciones".*

Desde el punto de vista de este principio **toda dependencia de algo concreto es volátil sobre todo al principio del diseño**. Aunque esta asunción puede ser algo radical , su observación prepara el camino ante posibles variaciones de determinados módulos.

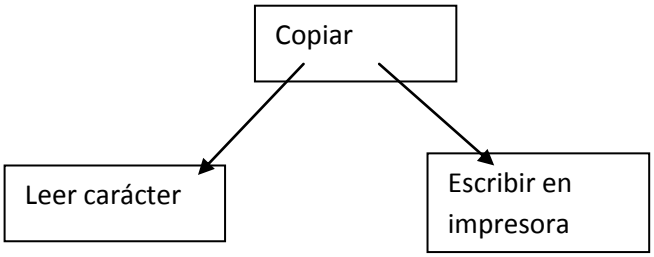
Otra motivación detrás de este principio podría ser que en un sistema generalmente se tienen clases de implementación de bajo nivel y clases de un nivel de abstracción más alto .Ocurre que las segundas normalmente son clases que encapsulan lógica de alto nivel haciendo uso de forma indirecta de las primeras. El objetivo es no acoplar dos niveles diferentes de abstracción

La aplicación de este principio p**romueve una estrategia basada en aplicar dependencias sobre interfaces y clases abstractas, en lugar de implementaciones concretas**. Este principio esta detrás del desarrollo basado en componentes como CORBA, COM y EJB.

Las **interfaces abstractas a nivel de diseño son normalmente más estables** (comparadas con sus implementación), así es más sencillo actualizar la implementaciones. Esto ayuda a incrementar las pruebas y decrementar la rigidez del diseño.

Como se vio en el estudio del principio abierto-cerrado para el diseño de una clase, las abstracciones son el lugar en el que es posible ampliar un diseño sin muchas dificultades. Cuanto más dependa un componente de otros componentes concretos (y no de abstracciones tales como una interfaz), más difícil será ampliarlo. Luego en definitiva tome el siguiente lema basado en este principio en consideración.***"Dependa de abstracciones. No dependa de implementaciones específicas".***

Vamos a ilustrar este principio con un ejemplo típico. Suponga que tenemos una parte del sistema que se encarga de copiar los caracteres leídos por teclado a una impresora. Este módulo tiene la siguiente estructura funcional:



Como vemos el modulo está gobernado por una función principal denominada Copiar que se apoya en otras dos: leer carácter y Escribir en impresora para desarrollar la función de copia

El código de esta función podría ser el siguiente:

```
void Copiar()
{
int c;
while ((c = LeerCaracterTeclado()) != EOF)
EscribirImpresora(c);
}
```

La función Copiar depende de que la fuente sea un teclado y el destino una impresora. En este ejemplo ficticio nos planteamos que sería interesante ampliar la capacidad de la función copia a cualquier contexto independientemente de la fuente y el destino. Esto dotaría a la función copia de una capacidad de reutilización considerable. Por ejemplo podríamos considerar que el modulo copiase caracteres a un archivo en el disco o a un socket.

Con el diseño actual podríamos modificar la función copia de la siguiente forma

```
void Copiar(outputDevice dev)
{
int c;
while ((c = LeerCaracterTeclado()) != EOF)
if (dev == printer)
EscribirImpresora(c);
else
EscribirDisco(c);
}
```

Se podría añadir una clausula “if” que seleccione entre los distintos destinos de la copia y en función del destino, realice la copia. **Esto funciona, pero añade nuevas dependencias**. A medida que el sistema evolucione podrían necesitarse nuevos dispositivos para copiar. El módulo de copia se dividiría en numerosas sentencias”if”, haciendo a este dependiente de cada dispositivo de bajo nivel. Con el tiempo este diseño se convierte en frágil y rígido ante posible cambios.

Una manera de caracterizar el problema anterior, es considerar que el módulo contiene una política de alto nivel, es decir realizar una copia y que es dependiente del tipo de origen y

destino de la copia a (es decir escribir en la impresora y escribir en disco). Si se pudiese encontrar una forma de hacer el módulo de copia independiente de estos detalles se podría reutilizar en cualquier en cualquier sistema que tuviera la necesidad de copiar desde cualquier dispositivo de entrada hacia cualquier tipo de dispositivo de salida.

Vamos a dar una solución basándonos en el principio de inversión.

Considera el diagrama de clases de la siguiente figura (Figura 13). El diagrama una clase Copia que contiene una clase de lectura abstracta y una clase de escritura abstracta. Es fácil imaginar un bucle donde la clase copiar obtenga los caracteres de la clase Lector y una vez obtenidos se los envié a la clase Escritor. Como se puede observar con este diseño la clase Copia no depende del teclado para leer ni de la impresora para escribir. **Estas dependencias se han invertido; la clase Copia depende ahora de abstracciones solamente.**

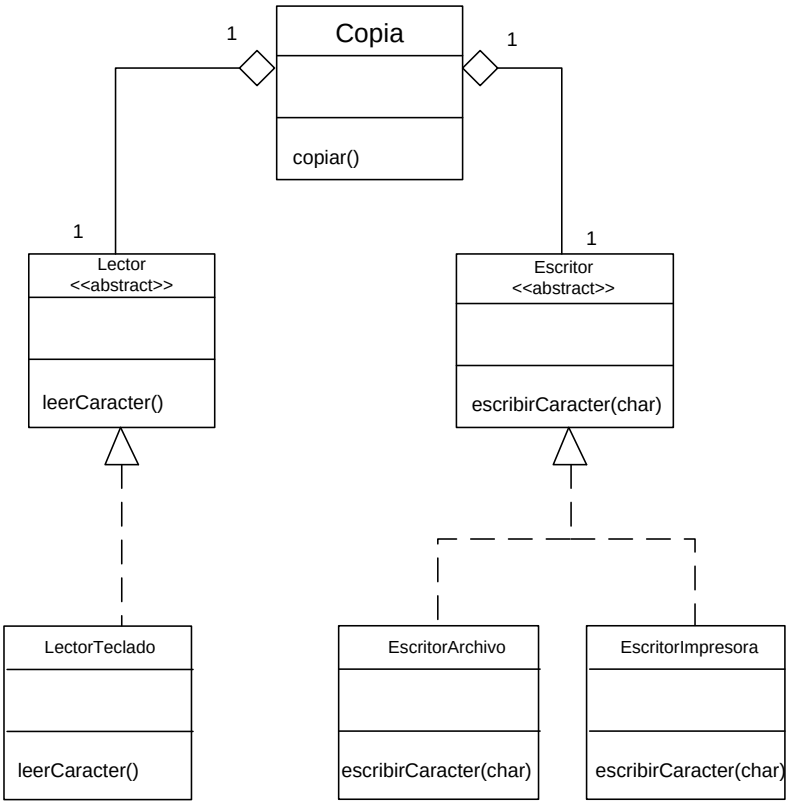


Figura 13 Solución empleando capas de abstracción

Ahora podemos reutilizar la **política de copia** en distintos contextos.

7. Convenciones de representación de componentes

Además de los principios estudiados anteriormente, conforme avanza el diseño a nivel de componentes se deberían aplicar y considerar una serie de planteamientos aplicados a los

componentes, a sus interfaces y a las características de herencia y dependencias que tengan algún efecto en el diseño resultante **Ambler[Amb02]** sugiere los siguientes.

- **Componentes.** Deben establecerse convenciones para dar nombre a los componentes que se especifiquen y que forman parte del modelo de la arquitectura, para luego mejorarlos y elaborarlos como parte del modelo de componentes. Los **nombres** de los componentes de arquitectura **deben partir del dominio del problema** y **significar algo para los participantes** que vean y analicen el modelo de arquitectura del sistema. En el siguiente ejemplo se muestran dos componentes que realizan la misma función pero con distintos nombres.

Por ejemplo el nombre **PlanoDeCasa para un componente** tiene un significado para todo aquel que lo lea, aunque no tengan conocimientos técnicos. Por otro lado los componentes de infraestructura y o clases elaboradas a nivel de componentes deben recibir un nombre que tengan un significado especifico en la implementación.

Pueden usarse estereotipos para ayudar a identificar la naturaleza de los componentes a nivel de diseño detallado. Por ejemplo el estereotipo **<<infraestructura>>** debiera usarse para identificar un componente de infraestructura, **<<base de datos>>** para un componente que de servicio a una o más clases de diseño

- **Interfaces.** Las interfaces proporcionan información importante sobre la comunicación y colaboración (también ayudan a cumplir el principio abierto-cerrado). Sin embargo la representación sin restricciones de las interfaces tiende a complicar los diagramas de componentes. **Ambler[Amb02c]** recomienda:

  1.- Si los diagramas aumentan en complejidad, en lugar del enfoque formal de UML con recuadro y flecha, debe representarse la interfaz de forma simplificada.

  2.-Las interfaces deben salir desde el lado izquierdo de la caja de representación del componente

  3.-Sólo deben aparecer aquellas interfaces que sean relevantes para el componente que se está considerando, aún si estuvieran disponibles otras.

Estas recomendaciones buscan simplificar la naturaleza visual de los diagramas UML de componentes

Dependencias y herencia. Para tener una mejor legibilidad, es buena indicar las dependencias de izquierda a derecha y la herencia de abajo (clases obtenidas) hacia arriba (clases base). Además, **las interdependencias de componentes deben representarse por medio de interfaces** y no con la dependencia componente a componente.

8. Realización del diseño de componentes

Especificación interna de las colaboraciones entre clases de objetos.

Se pueden usar **diagrama de colaboración y secuencia UML** para mostrar como colaboran los objetos instancia de las clases internas del componente para realizar la funcionalidad descrita por la interfaz del componente. A medida que avanze el diseño es necesario especificar más en detalle estas interacciones por ejemplo indicando la **estructura de los mensajes** (número de argumentos y el tipo)y secuencia temporal de los mismos.

La siguiente figura (Figura 16) ilustra un diagrama sencillo de colaboración entre clases de objetos para el sistema de impresión.

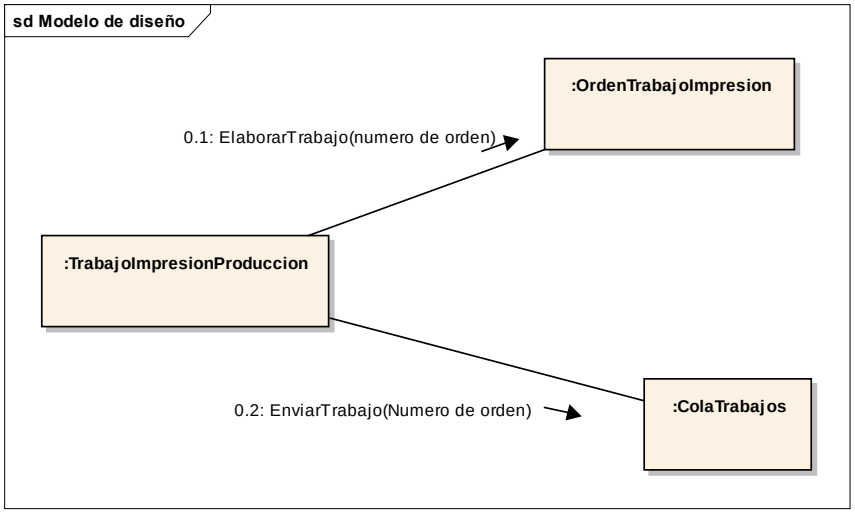


Figura 14 Diagrama de comunicación

Los objetos TrabajoImpresionProducción, OrdenDeTrabajoImpresion y ColaTrabajos colaboran para preparar un trabajo de impresión a fin de iniciar su producción. A medida que avanza el diseño estos mensajes se irán detallando incluyendo su estructura completa que como regla general tiene el siguiente formato

[condición] secuencia (valor retorno):= nombre mensaje(lista argumentos)

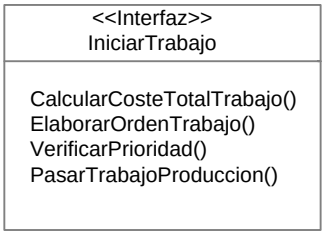
Donde [**condición]** especifica cualquier condición que deben cumplirse **antes de enviar el mensaje**, **secuencia** indica es un valor entero que indica el orden secuencial del mensaje dentro del contexto de la colaboración, por otro lado tenemos además el **valor de retorno** que devuelve la invocación del mensaje y por último la **lista de argumentos** que se envían en la invocación.

Identificar las interfaces apropiadas para componente

En el contexto del diseño de componentes, una interfaz como hemos visto es un "grupo de operaciones accesibles desde el exterior .La interfaz no contiene estructura interna, ni atributos ni asociaciones "[Ben02]. En la Figura 1 se ha ilustrado la elaboración de interfaces.

En esencia, las operaciones definidas para cada clase de diseño se clasifican en una o más clases abstractas. **Cada operación** dentro de la clase abstracta (la interfaz) **debe ser cohesiva.**

En relación con la **Figura 1**, puede afirmarse que la interfaz **IniciarTrabajo no tiene suficiente cohesión**. En realidad **ejecuta tres funciones** diferentes: elaborar una orden de trabajo de impresión, verificar la prioridad del trabajo y pasar el trabajo a producción.



La interfaz debería rediseñarse. Un nuevo enfoque podría ser el estudiar las clases de diseño y definir una clase nueva denominada **OrdenTrabajoImpresión,** que se centraría en las actividades asociadas a la formación del trabajo de impresión. La operación **elaborarOrdenTrabajo()** seria implementada por esta clase. De forma similar, se podría definir una clase denominada **ColaTrabajos** que implementaría el servicio de v**erificarPrioridad()** .Una clase **TrabajoDeProduccion** podría incorporar toda la información asociada con un trabajo que pasará a las instalaciones de producción. Siguiendo el principio de segregación dividimos la interfaz **IniciarTrabajo** en las interfaces **TrabajoDeProduccion, OrdenTrabajo y ColaTrabajos** de forma que estás interfaces **ahora si son cohesivas.**

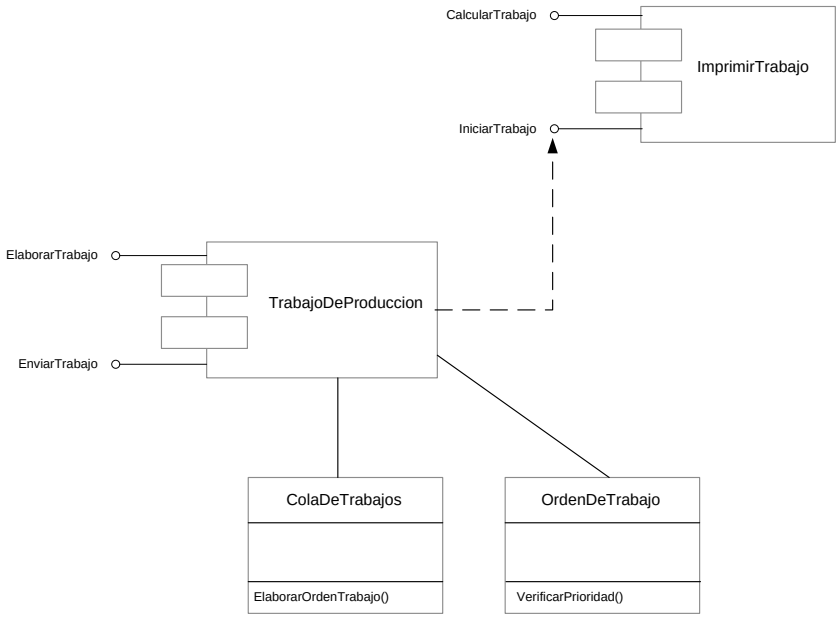


Ilustración 2 detalle de implementación de interfaces

Elaborar los atributos, definir tipos de datos y estructuras para representarlos

En general esto debe **hacerse teniendo en cuenta el contexto donde se va implementar el componte**, nos estamos refiriendo a la plataforma de ejecución, componentes existentes y lenguaje de programación.

Durante la primera iteración de diseño de componentes los atributos de las clases se indican con el nombre simplemente ver Figura 1. No obstante a medida que avanza el diseño hay que definir qué tipos de datos se van a utilizar para representar estos atributos y si estos son complejos que clases los va a representar.

Describir el flujo de proceso del componente

Esta tarea se puede realizar o bien en pseudocódigo o con un **diagrama de actividades UML**. A través de estas representaciones se elabora cada operación buscando un nivel de cohesión alto y un alto nivel de comprensión de cómo se realiza la operación.

Poe ejemplo para la operación **CalcularCostePapel** del ejemplo se define de la siguiente forma

```
CalcularCostePapel ( peso,tamaño, color): numérico
```

Que indica que la operación requiere tres parámetros y como valor de retorno obtenemos un valor numérico que indica el valor del coste.

Si el algoritmo de cálculo del coste del papel es sencillo y entendido por todos los miembros del equipo, tal vez no sea necesaria mucha elaboración. Sin embargo si el algoritmo es complejo o difícil de entender, sería conveniente realizar una representación para clarificar este algoritmo. La siguiente figura **(Figura 12)** muestra un diagrama de actividades UML para especificar el diseño. Por lo general el nivel de abstracción de estos diagramas es alto, si se compara con la posterior implementación en un lenguaje. Pero son muy útiles para detallar y estructurar las actividades realizadas en alguna función compleja. Estos diagramas muestran el comportamiento general de la operación.

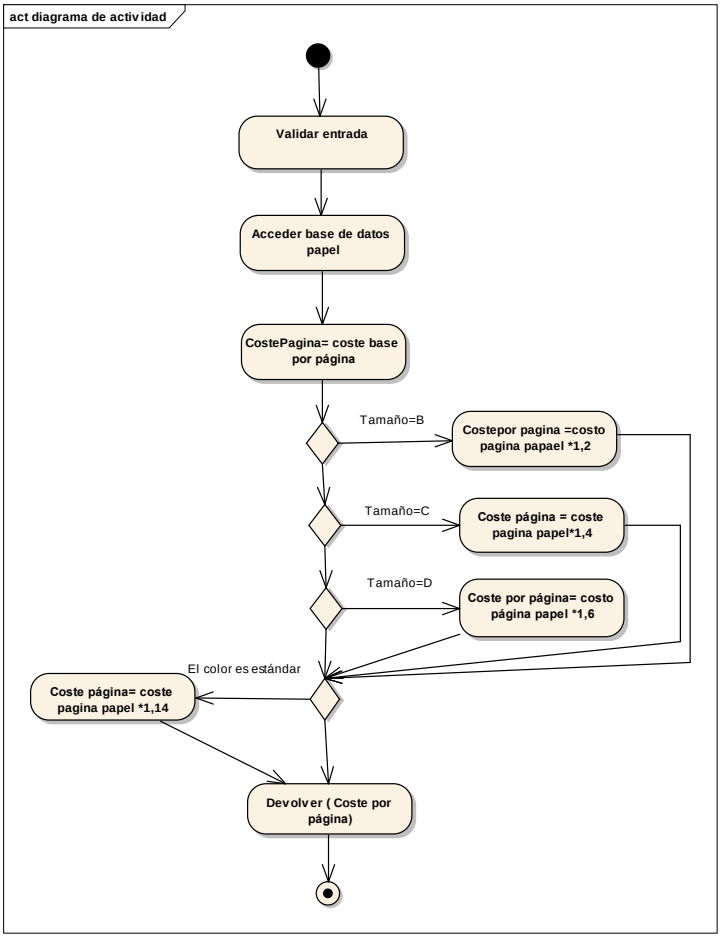
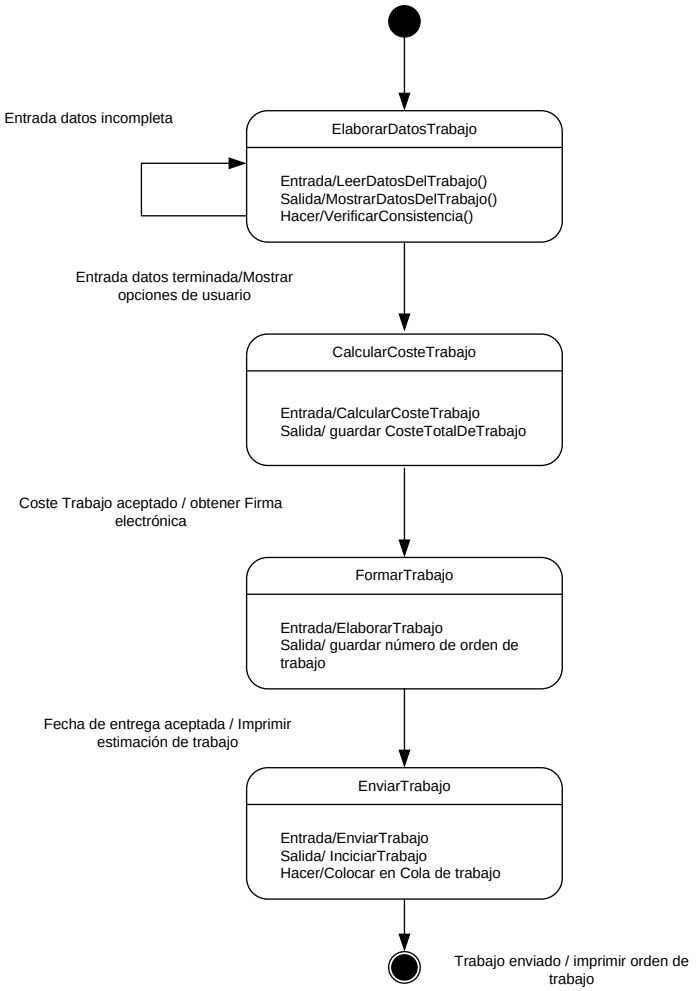


Figura 15 Diagrama de actividad para la operación **CalcularCostePapel**

Representación del comportamiento de un componente.

Durante el diseño, **en ocasiones es necesario modelar el comportamiento dinámico de una clase de objetos o de un componente**. El comportamiento dinámico de un objeto se ve afectado por eventos externos a él y por el estado en curso del propio objeto. Para entender el comportamiento dinámico de un objeto, deben estudiarse todos los estados y usos que sean relevantes. Los casos de uso dan información que ayuda a delimitar los eventos que afectan al estado de un objeto. Las transiciones entre estados (determinadas por eventos) se representan con un **diagrama de estados** como el que se muestra en la siguiente figura.



La transición a un estado (representada por un cuadrado con las esquinas redondeadas) sucede por la ocurrencia de algún evento, cuya sintaxis es la siguiente:

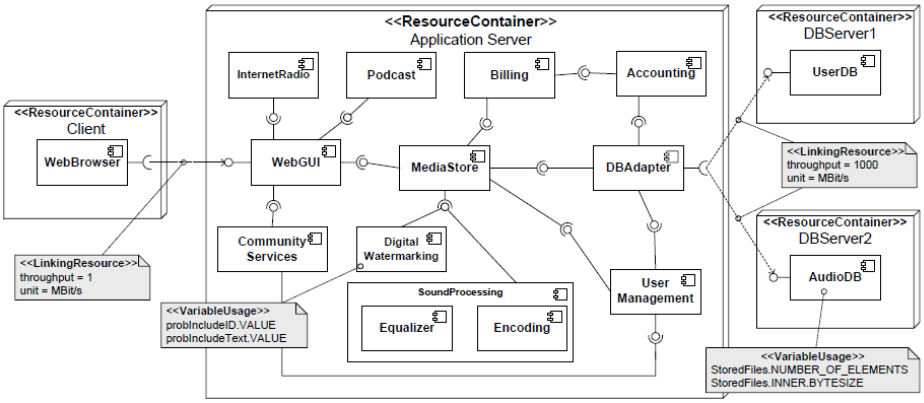
```
Nombre_evento( parámetros) [condición de guarda]/ acción
```

Donde nombre del evento indica el evento que produce una transición, parámetros indica la lista de argumentos asociados al evento

Diagramas de despliegue para dar detalles de la implantación

Los diagramas de despliegue se usan como parte del diseño de la arquitectura física. Las funciones de un sistema que frecuentemente se representan como subsistemas se representan en el contexto de entorno hardware donde van a ir instaladas.

En el diseño de componentes, se pueden elaborar diagramas d despliegue para representar la ubicación física de cada componente. A continuación se muestra un diagrama de despliegue para un sistema de gestión de contenidos digitales.



9. Bibliográfica

- Roger S. Pressman .Ingeniería de software un enfoque práctico, Mc Graw hill
- Meyer, B. *Object-Oriented Software Construction*. Englewood Cliffs, N.J. Prentice-Hall, 1988.
- Grady Booch, Object Solutions, Addison Wesley 1996
- The original MVC reports Trygve Reenskaug Dept. of Informatics University of Oslo
- Design Principles and Design Patterns Robert C. Martin
- Clemens Szyperski Component Software - Beyond Object-Oriented Programming Addison-Wesley / ACM Press, 1998 (411 pages)
  ISBN 0-201-17888-5