Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Resúmen

Ingeniería de Software II

Autor:
Arroyo, Joaquín

February 23, 2024

Contents

1 Introducción a Diseño

1.1 ¿Por qué diseñar software?

Definición 1 (Diseño de Software). Diseñar un sistema de software significa:

1. *Descomponerlo o dividirlo en elementos de software. Es decir, pensar al software como un conjunto de elementos que interactúan entre sí, y no como un bloque indiviso.*
2. *Asignar y describir una función para cada uno de esos elementos.*
3. *Establecer las relaciones entre esos elementos. Las relaciones entre los elementos representan las posibles interacciones entre ellos. Un elemento de software puede ser referenciado, usado o accedido por otros elementos de muchas formas diferentes. Cada una de estas posibilidades es una relación entre elementos de software.*

Es conveniente dedicar tiempo y esfuerzo en diseñar un sistema de software porque es más barato desarrollarlo y mantenerlo, que hacerlo como se lo hace habitualmente.

La producción de un sistema de software se puede dividir en dos grandes etapas: desarrollo y mantenimiento. Durante el desarrollo el programa se escribe desde cero (o casi si se utilizan porciones de código previamente desarrolladas) hasta que se entrega al cliente la primera versión. Durante el mantenimiento se desarrollan nuevas versiones hasta que el sistema se vuelve obsoleto y es descartado. Se estima que en promedio el desarrollo ocupa el 33% del esfuerzo total de producción en tanto que el 67% restante se dedica al mantenimiento, aunque diversos autores y estudios difieren en las cantidades exactas.

El mantenimiento consiste esencialmente en cambiar, agregar o eliminar líneas de código fuente al software tal y como está en cada momento. Esto se hace tanto para corregir, adaptar o mejorar la versión actual. En resumen, gran parte del costo asociado con la producción de un sistema de software proviene de la necesidad de incorporar cambios a una versión del sistema (antes o después de la entrega) con el fin de:

- Cambiar, agregar o eliminar código fuente.
- Corregir, adaptar, mejorar.
- Elaborar una línea de productos.
- Agregar requerimientos tardíos.
- Modificar requerimientos erróneos.

Dentro de los costos originados por la incorporación de cambios, el más alto es el ocasionado por la necesidad de tener que volver a testear todo o una gran parte del sistema cada vez que se introduce un cambio.

Claramente, entonces, los cambios no son sólo inevitables o inherentes a la producción de software sino que en muchas ocasiones son un excelente negocio. Por lo tanto no tiene sentido buscar estrategias para reducir la cantidad de cambios. Por el contrario, se deben buscar principios, metodologías, técnicas y herramientas de forma tal de cumplir con dos objetivos básicos de diseño:

- Incorporar cada cambio con el menor costo posible.
- Evitar que cada cambio degrade la integridad conceptual del sistema Si esto no se tiene en cuenta, incorporar los primeros cambios puede ser barato pero cada vez será más costoso.

Es imposible lograr los objetivos anteriores sin haber tenido en cuenta los posibles cambios que sobrevendrán. En consecuencia se deben analizar los cambios posibles, la línea evolutiva del sistema, los productos que de él pueden derivarse, etc. Dado que introducir cambios tiene que ver con modificar líneas de código fuente, entonces es fundamental descomponer, dividir u organizar bien el código fuente, es decir, diseñarlo bien. Por lo tanto, dedicar esfuerzo para diseñar el sistema resulta más económico que no hacerlo porque permitirá que los costos de mantenimiento, e incluso parte del costo de desarrollo, disminuyan sensiblemente dado que se trata fundamentalmente de introducir cambios. Y como el costo

de mantenimiento es por mucho el mayor de todos, entonces cualquier estrategia que tienda a reducirlo es esencial a la producción de software.

El diseño tiende a reducir los costos de mantenimiento porque reduce el costo de cambio, que es laprincipal fuente de costos, en consecuencia es una actividad esencial a la producción de software.

1.1.1 Diseño para el cambio

Uno de los principios de la Ingeniería de Software es el principio de Diseño para el Cambio. Este principio sugiere anticipar en el diseño del sistema los cambios que probablemente se quieran incorporar en el futuro. Simplemente dice que hay que pensar y poner esfuerzo en anticipar los cambios más probables y hacer algo para reducir su impacto en el momento en que haya que incorporarlos al sistema. No dice cómo lograrlo, pero si lo dijera no sería un principio. Claramente, de todo lo antedicho se deduce que cualquiera sea la teoría de diseño que se elija, esta debe basarse en el principio de Diseño para el Cambio.

1.2 Calcular diseño

Uno de los aspectos fundamentales de cualquier teoría de diseño de software es que los ingenieros deben tener claro que un diseño tiene una componente importante de intuición, experiencia y creatividad, pero que tiene una componente más importante de cálculo. Es decir, el diseño debe ser fruto de alguna regla de cálculo. Por eso se habla de diseño calculado.

Una de los objetivos esenciales de cualquier regla de cálculo para el diseño de software debe ser estipular claramente el criterio por el cual el sistema debe descomponerse en 2, 3 o 534 elementos de software, por qué tienen que ser esos elementos y no otros, por qué esos elementos tienen que tener tales y cuales funciones y no otras, porqué esos elementos se tienen que relacionar de tales o cuales maneras y no de otras. El ingeniero se enfrenta entonces al dilema de determinar no solo el número correcto de elementos en que debe descomponer el sistema sino las características estructurales fundamentales de esos elementos. Esto debería hacerse de una manera más o menos objetiva siguiendo una serie de pasos claros y que siempre lleven a un diseño, sino excelente, muy bueno.

Este criterio, llamado criterio de descomposición o criterio de diseño, es el núcleo de cualquier teoría de diseño de software, es la regla de cálculo, es la herramienta básica de cualquier ingeniero de software que trabaje en diseño. El criterio de diseño le da al ingeniero una regla por medio de la cual hacer la misma cosa de dos formas diferentes:

1. Determinar los elementos de software que debe tener su diseño y las características fundamentales de esos elementos (interfaces, relaciones, especificaciones, etc.).

2. Verificar si un diseño dado es correcto o no, puesto que puede analizar si ese diseño es el resultado de haber aplicado la regla.

Cualquier criterio de diseño debe someterse al principio de Diseño para el Cambio de forma tal de cumplir los dos objetivos básicos de diseño, porque de lo contrario ese criterio no estaría teniendo en cuenta la justificación industrial que sustenta la necesidad de diseñar el software.

1.3 El diseño como fase del ciclo de desarrollo del sistema

El diseño de software forma parte de la fase del ciclo de desarrollo del sistema denominada Arquitectura del Sistema. Considerando el modelo de desarrollo de cascada, la fase de definición de la arquitectura se ejecuta luego de la Ingeniería de Requerimientos. Es decir que después de contar con una lista más o menos estable de requerimientos, los arquitectos del sistema están en condiciones de pensar, concebir y documentar una arquitectura para el sistema.

1.3.1 Diseño y métodos formales

Un modelo funcional describe los requerimientos del usuario mediante algún formalismo lógico o matemático. Por lo tanto, el diseño de un sistema y su modelo funcional son dos descripciones diferentes. En otras palabras son dos formas diferentes pero complementarias de ver un sistema.

En efecto, dado un cierto conjunto de requerimientos, un sistema que los satisfaga funcionalmente puede haber sido diseñado de varias formas diferentes. Un modelo funcional puede ser estructurado de diferentes formas; el modelo funcional no fija necesariamente una única forma de descomponer el sistema. Más aun, en la mayoría de los casos es un error tomar el modelo funcional como diseño del sistema.

Pero entonces, ¿dónde encajan en el diseño los lenguajes formales para especificación que vimos anteriormente? Básicamente se trata de que la función deseada de cada elemento del diseño se especifique con uno o varios de esos lenguajes. En efecto, la función que se le asigna a cada elemento del diseño es una porción de los requerimientos o ayuda a implementar parte de los requerimientos.

De todas formas, no siempre es necesario o económicamente viable o conveniente especificar formalmente la función de todos los elementos en que se descompone un sistema. Justamente, el hecho de primero diseñar (descomponer) y luego especificar, permite que antes de especificar se seleccionen las porciones más complejas, críticas o convenientes con el fin de ser especificadas formalmente. A las restantes se les adosará una especificación informal.

1.4 Los tres niveles estructurales

La estructura de un sistema de software se realiza o define en tres niveles de abstracción diferentes, listados a continuación desde el más abstracto al más detallado:

1. Elección del estilo arquitectónico. Cada estilo arquitectónico representa a una familia de sistemas de software que comparten características estructurales fundamentales. Mediante un estilo arquitectónico se puede definir a grandes rasgos la estructura de un sistema de software de dimensión industrial. Existe alrededor de una docena de estilos arquitectónicos.

2. Selección de los patrones de diseño. Cada patrón de diseño representa una solución general reusable para un problema que se da con frecuencia en el diseño de software. No es posible en general definir la estructura completa de un software usando uno o dos patrones de diseño. Existen tal vez más de 200 patrones de diseño.

3. Diseño de componentes.

Cuando un ingeniero de software aborda la definición de la estructura de un sistema debe comenzar por trabajar en el más alto nivel de abstracción para luego ir refinando dicha estructura incorporando más detalles. Usualmente la primera decisión que debe tomar el ingeniero es el o los estilos arquitectónicos en los que se basará la estructura del sistema, la función que tendrán los componentes principales de aquellos y sus relaciones. Luego podrá seleccionar patrones de diseño para finalizar definiendo los componentes de menor nivel de abstracción o dando más detalles sobre los componentes ya definidos. Es muy común que se deban realizar varias iteraciones entre los niveles 2 y 3; si bien no debería ser frecuente, puede ocurrir que el estilo arquitectónico deba ser modificado. Esto último puede darse cuando los requerimientos desde los que parte el arquitecto tienen un nivel de calidad muy pobre. Aun teniendo que alterar la primera decisión estructural, es siempre significativamente más barato recalcular o redefinir el diseño que tener que hacerlo una vez que hay grandes porciones de código ya escritas. Por consiguiente, el equipo de arquitectos debe disponer de un tiempo razonable para describir la arquitectura con un nivel de detalle suficiente como para tener cierta confianza de que los cambios más probables podrán ser incorporados a bajo costo y sin degradar la integridad conceptual del sistema.

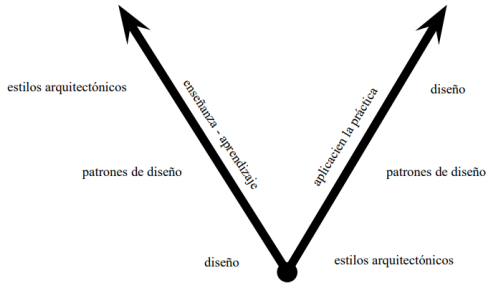


Figure 1: Las etapas de la fase Arquitectura del Sistema se enseñan-aprenden de forma opuesta a la forma en que se aplican en la práctica.

1.5 Ciclo de vida de la arquitectura

En esta sección veremos cómo la arquitectura del sistema afecta a su entorno y este también afecta a la arquitectura.

Los interesados en una arquitectura de software de un cierto sistema son todas aquellas personas u organizaciones que tienen alguna injerencia o interés en el sistema. Nunca se repetirá la cantidad de veces suficiente lo importante que es determinar con la mayor precisión posible el conjunto de interesados en el sistema antes de comenzar a delinear su arquitectura. Determinar el conjunto de interesados en el sistema es responsabilidad o injerencia de los ingenieros a cargo de la Ingeniería de Requerimientos, como primer paso antes de comenzar con la captura de los requerimientos. Un interesado relevante que no sea consultado tempranamente sobre la arquitectura del sistema, será una fuente de problemas en el futuro.

La arquitectura de software de un sistema es el resultado de combinar decisiones técnicas, sociales y del negocio. Los ingenieros de software deben convencerse que su trabajo está influenciado por el negocio y el entorno social que los rodea. Es ilusorio e ingenuo suponer que uno podrá determinar la arquitectura únicamente basándose en consideraciones técnicas. Los interesados, algunos de ellos más que otros, más tarde o más temprano presionarán al arquitecto del sistema para que la arquitectura tenga tal o cual característica que no necesariamente es la mejor desde el punto de vista técnico. Los conocimientos técnicos del arquitecto así como también el estado del arte de la práctica profesional de la Ingeniería de Software también son fuertes condicionantes no técnicos a la hora de definir la arquitectura de un sistema. Si el arquitecto domina el DOO entonces sus arquitecturas tenderán a seguir esa técnica; si se está ideando un sistema de comercio electrónico, la industria y la cultura técnica imperantes presionan para utilizar webservices independientemente de que sea la tecnología óptima para el sistema en cuestión.

El ciclo ABC se cierra pues una arquitectura exitosa tenderá a convertirse en la referencia obligada dentro de la organización para estructurar sistemas semejantes o elaborar líneas de productos. Asimismo los usuarios finales o los clientes serán renuentes a definir una nueva arquitectura para un nuevo sistema si la anterior fue la base para un sistema que les brindó buenas prestaciones. Finalmente, ciertos sistemas basados en ciertas arquitecturas logran rebasar las fronteras de una organización y se convierten en estándares de-facto o referencias obligadas de la industria por lo que terminan teniendo una influencia importante sobre la práctica profesional.

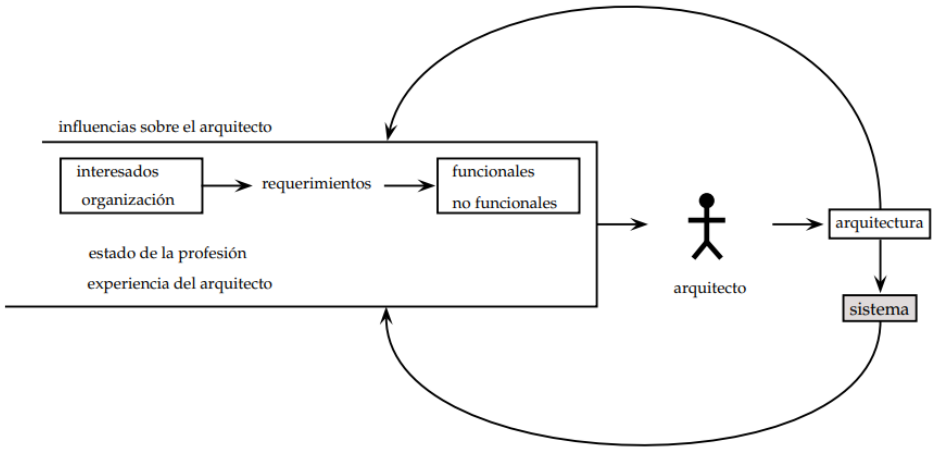


Figure 2: Influencias mutuas entre la arquitectura y su entorno.

1.5.1 La influencia de los requerimientos no funcionales en la definición de la arquitectura de un sistema

En la **Figura 2** se presenta a los requerimientos como una influencia determinante en la concepción de la arquitectura de software del sistema. Más aun los requerimientos se dividen en dos grandes clases: funcionales y no funcionales. Normalmente los requerimientos funcionales no ocupan solo la primera posición sino la única a la hora de definir la arquitectura del sistema. Usualmente requerimientos

no funcionales o cualidades tales como modificabilidad, seguridad, desempeño, tolerancia a fallas, testeabilidad, etc. no son tenidas en cuenta ni por los arquitectos del sistema ni por la mayoría o todos los interesados. En general la necesidad de que el sistema verifique algunas de estas cualidades se percibe recién cuando este entra en producción o en las etapas finales del desarrollo.

El problema de esta situación radica en que por lo general lograr que un sistema casi terminado tenga un desempeño superior, sea modificable, seguro o tolerante a fallas es casi imposible si no se pensó desde su concepción, es decir si esas cualidades no se incorporaron conscientemente en la arquitectura del sistema. No ocurre lo mismo, en la mayoría de los casos, con el requerimiento de agregar nuevas funcionalidades; suele ser costoso pero en general no requiere rehacer el sistema por completo. Los requerimientos funcionales por lo general son discretos en el sentido de que agregando o modificando algunas líneas de código en unos pocos lugares es suficiente para implementarlos; mientras que los requerimientos no funcionales son, por lo común, densos en el sentido de que es necesario agregar o modificar código en todas partes para implementarlos. En consecuencia no prever un requerimiento no funcional suele ser mucho más costoso que no tener en cuenta un requisito funcional. Actualmente se considera que los requerimientos no funcionales deben guiar la definición de la arquitectura del sistema tanto como los funcionales.

1.6 La diferencia entre arquitectura y diseño

Definición 2 (Nivel Arquitectónico). *El nivel arquitectónico de la estructura de un sistema es aquella descripción donde se utilizan conectores diferentes a llamada a procedimiento y/o se imponen restricciones estructurales importantes entre los componentes y/o se utilizan distintos tipos de componentes.*

Antes de analizar la definición clarificaremos algunos conceptos que en ella aparecen:

Componente: Entidad computacional activa que implementa algún requisito funcional o parte de este.

Conector: Mecanismo que mediatiza la comunicación, coordinación o cooperación entre componentes.

Tipo componente: Componentes que comparten características estructurales, en particular, y fundamentalmente, los mismos tipos de interfaz.

Tipo interfaz: Forma de interacción con el entorno semántica y estructuralmente única.

En primer lugar notar que la definición no habla de la arquitectura del sistema sino de una descripción particular de la estructura, por lo que decimos que esta definición refina o complementa a la **Definición 1**. Observar que la definición refiere a las mismas actividades que se llevan a cabo en el diseño solo que en un nivel de abstracción diferente. En este sentido, la arquitectura es diseño.

La diferencia distintiva con respecto al nivel del diseño radica en que según nuestra definición la descripción del nivel arquitectónico implica el uso de elementos de software que no tienen una representación directa en la mayoría de los lenguajes de programación; es decir, elementos abstractos con los cuales trabaja el arquitecto y que los programadores deberán refinar y proyectar sobre la tecnología de implementación disponible. En el nivel arquitectónico (no en el del diseño) se debe hacer hincapié en componentes con interfaces complejas y/o con relaciones complejas entre ellos y, primordialmente, conectores semánticamente más ricos. . Considerar a los conectores como elementos de primer nivel implica entender que una buena arquitectura depende tanto de las relaciones entre sus componentes como de los componentes en sí. Más aun, dado que los conectores en el nivel arquitectónico son abstracciones no directamente representables en el lenguaje de programación, suele ser muy común que estos se conviertan en componentes al ser descriptos en los niveles inferiores.

También podemos determinar si una decisión estructural es arquitectónica o no por medio del Criterio de Localía, el cual dice que una decisión estructural no es de diseño (y por lo tanto es arquitectónica) sí y sólo sí un sistema que verifica esa decisión puede ser extendido a un sistema que no la verifica. Por ejemplo, estructurar un sistema siguiendo el estilo cliente/servidor es una decisión arquitectónica puesto que el sistema puede ser extendido a un sistema peer-to-peer que no verifica el estilo. Por otro lado, si ese mismo sistema utiliza, por ejemplo, el patrón de diseño Strategy, al extenderlo a un sistema peer-to-peer seguirá utilizándolo, por lo que el uso del patrón es una decisión de diseño (y no arquitectónica).

Sin embargo, no siempre es simple diferenciar claramente entre el nivel de diseño y el nivel arquitectónico. En última instancia es el ingeniero el que traza la línea que divide ambos niveles en

cada proyecto. Precisamente, no disponemos aun de una notación que permita expresar siempre el nivel arquitectónico sin adentrarse en el detalle del diseño. Sin embargo, ciertas estructuras permiten describir este nivel, o son más útiles en ese nivel que en el nivel del diseño. Una buena aproximación para describir la arquitectura sin describir el diseño yace en el concepto de estilos arquitectónicos.

1.7 Estilos arquitectónicos

Los estilos arquitectónicos son una generalización y abstracción de los patrones de diseño.

Definición 3 (Estilo Arquitectónico). *Caracteriza una familia de sistemas que están relacionados por compartir propiedades estructurales y funcionales.*

También puede definirse como la descripción de los tipos componente y de los patrones de interacción entre ellos.

Notar que, a diferencia de los patrones de diseño, la definición apunta a describir sistemas completos y no partes de sistemas. Nadie supone que podrá describir un sistema completo mediante el patrón de diseño Composite o Abstract Factory o Command, pero sí puede hacerlo mediante un estilo arquitectónico.

2 Diseño basado en ocultación de la información (DBOI)

El DBOI se basa en uno de los principios de la Ingeniería de Software: Diseño para el Cambio. Es decir, no suponer jamás que los requerimientos de un sistema serán dados de una vez y para siempre sino que cambiarán muchas veces durante la vida del sistema. Por lo tanto, se debe concebir al sistema de forma tal que:

- Pueda ir incorporando los nuevos requerimientos evitando que su integridad conceptual se degrade.
- Los cambios puedan incorporarse con el menor costo posible.

Resulta evidente, entonces, que antes de comenzar con el diseño propiamente dicho se deben analizar las posibles líneas evolutivas del sistema y los cambios probables en los requerimientos. Por otro lado, cada vez que debamos introducir un cambio deberemos modificar una o más unidades de implementación de software (código fuente, archivos de configuración, etc.). Cuando una unidad de software implementa una funcionalidad específica se la llama módulo.

Definición 4. *Un módulo es una unidad de implementación de software que provee una unidad coherente de funcionalidad. También se puede definir como una unidad de implementación de software que provee un conjunto de servicios. Parnas define módulo como una asignación de trabajo para un programador o un grupo de programadores.*

Como el Diseño para el Cambio sugiere tener en cuenta los cambios probables para incorporarlos con el menor costo posible, y los cambios se deben realizar dentro de los módulos del sistema, entonces se busca descomponer el sistema en módulos (es decir, diseñar los módulos) de forma tal que:

- Cada módulo se pueda implementar independientemente de los restantes.
- Cada módulo pueda ser comprendido completamente sin necesidad de comprender los otros en su totalidad.
- Sea posible cambiar la implementación de un módulo sin conocer la implementación de los otros y sin afectarlos.
- Sea posible incorporar un cambio importante como un conjunto de cambios pequeños a distintas módulos.

2.1 La metodología de Parnas para descomponer un sistema en módulos

Al descomponer un sistema en módulos se alcanzan los objetivos anteriores si se siguen los siguientes pasos:

1. Se identifican los ítem con probabilidad de cambio presentes en los requerimientos.
2. Se analizan la diversas formas en que cada ítem puede cambiar.
3. Se asigna una probabilidad de cambio a cada variación analizada.
4. Se aislan en módulos separados los ítem cuya probabilidad de cambio sea alta; implícitamente este punto indica que en cada módulo se debe aislar un único ítem con probabilidad de cambio.
5. Se diseñan las interfaces de los módulos de manera que resulten insensibles a los cambios anticipados.

A esta forma de diseñar se la denomina diseño basado en ocultación de información. Obviamente, la información que se oculta es la implementación de cada ítem que probablemente cambiará en el futuro. En otras palabras: cada módulo de la descomposición se caracteriza por su conocimiento de una decisión de diseño que oculta a los demás módulos; su interfaz se elige de manera tal de revelar lo menos posible sobre su maquinaria interna.

Principio de Ocultación de la Información (POI): Los ítem con alta probabilidad de cambio son el fundamento para descomponer un sistema en módulos. Cada módulo de la descomposición debe ocultar un único ítem con alta probabilidad de cambio, y debe ofrecer a los demás módulos una interfaz insensible a los cambios anticipados.

Principio de diseños Abiertos y Cerrados El diseño debe ser tal que nuevas funcionalidades se puedan incorporar en nuevos módulos sin tener que modificar los módulos existentes. Los módulos de un diseño deben estar abiertos a extensiones, pero cerrados a modificaciones.

Definición 5 (DBOI). *Un sistema respeta el diseño basado en ocultación de la información (DBOI) si cada uno de sus módulos fue diseñado aplicando el POI y se siguieron adecuadamente cada uno de los pasos anteriores.*

2.2 Análisis de cambio

Par realizar esta tarea se tienen en cuenta los siguientes puntos:

- Contracción y extensión de los requisitos (es decir pensar que algunos clientes pueden querer un sistema más grande, complejo o completo del que se está diseñanado mientras que otros pueden querer un sistema más pequeño, más simple, con menos funciones, etc.).
- Cambio de/en los algoritmos.
- Cambio en la representación de las estructuras de datos y en la forma de organizarlos
- Cambio en la máquina abstracta subyacente (hardware, sistema operativo, compilador, runtime, etc.).
- Cambio en los dispositivos periféricos
- Cambio en el entorno socio-cultural (moneda, impuestos, fechas, idioma, etc.).
- Cambios propios del dominio de aplicación.
- Cambios propios del negocio de la compañía desarrolladora Interconexión con otros sistemas.

2.3 Interfaz e implementación

Para entender los pasos 4 y 5 de la metodología de Parnas, es necesario entender primero qué forma o propiedades tienen los módulos. Un módulo puede ser visto, usado o accedido por otros módulos del sistema u otros sistemas. Un módulo de un sistema de software consta de dos secciones: interfaz e implementación. La interfaz de un módulo es todo aquello que los otros módulos pueden ver, usar o acceder en ese módulo. La implementación es la forma en que la interfaz se realiza; muchas veces se habla de secreto del módulo. Si un módulo puede ver, usar o acceder un elemento en otro módulo es porque este último exportó ese elemento. O sea que la interfaz de un módulo es todo lo que el módulo exporta.

Definición 6. *La interfaz de un módulo puede definirse como el conjunto de servicios que el módulo exporta. También puede definirse como todas las interacciones que tiene el módulo con su entorno (es decir, los otros módulos).*

Algunos de los servicios que un módulo puede exportar son: declaraciones de tipos, subrutinas, variables, constantes, etc.

Algunas de las interacciones con el entorno son: llamadas a subrutinas, tiempo de ejecución de esas subrutinas, especificación funcional de esas subrutinas, invariantes que preserva el módulo, etc.

Si un servicio pertenece a la interfaz de un módulo se dice que el servicio es público.

Definición 7. *La implementación de un módulo es la forma en que se logra que la interfaz funcione según lo perciben los otros módulos. Cualquier elemento de la implementación se dice que es privado.*

Claramente la interfaz de un módulo captura la visión externa del módulo. Cuando un módulo presenta una interfaz tal que los otros módulos no pueden suponer razonablemente nada sobre la implementación del módulo, se dice que la implementación está **encapsulada**.

2.4 Abtracción y encapsulamiento

La *abstracción* consiste en lograr que la interfaz provea la menor cantidad de servicios posible y de la manera más abstracta posible. En tanto que el *encapsulamiento* es el proceso por el cual se ocultan todos los detalles de la implementación que permanecen visibles en los servicios exportados.

Definición 8. Interfaz Fina. *La forma básica de aplicar la abstracción al diseño de un módulo es definir la interfaz como un conjunto de subrutinas (que, por definición, son lo único que otros módulos pueden ver, usar o acceder) y luego remover todas aquellas subrutinas que pueden ser realizadas por otras.*

Tener en la interfaz subrutinas que puede ser realizadas por otras da lugar a lo que se conoce como interfaces **gruesas**.

Finalmente, mediante el *encapsulamiento* de la interfaz se logran remover los últimos detalles de la implementación que permanecen visibles desde el exterior del módulo. Normalmente es necesario revisar la especificación funcional de cada subrutina (recordar que esto también forma parte de la interfaz) y los parámetros de cada una de ellas. Por un lado se busca que la especificación funcional de cada subrutina esté lo más cerca posible de los requerimientos funcionales y no implique, razonablemente, ninguna implementación particular. Por el otro, se intenta eliminar restricciones propias de una implementación particular que afloran en la interfaz.

2.5 Introducción a 2MIL

Para describir (documentar) las interfaces de los módulos usaremos un lenguaje muy simple llamado 2MIL.

Module	**Prueba**
imports	M_1, ..., M_n
exportsproc	proc1(**i** X)
	proc2():X
private	privproc()
comments	comentarios.

Los métodos tienen la siguiente sintaxis:

metodo(**i Type1**):**Type2**

Esto significa que el método recibe un parámetro de tipo $Type1$ y devuelve algo de tipo $Type2$. Pueden existir métodos que no reciban nada y/o que no devuelvan nada. Pueden existir métodos que reciban mas de un parámetro. Notar que **i** no es el nombre del parámetro de entrada, si no que indica que es un *input*.

2.6 Superioridad y limitaciones de DBOI

Existe una diferencia metodológica muy importante entre el DBOI y el DOO como se lo define habitualmente. Según la mayor parte de la literatura dedicada al DOO los módulos se determinan por la existencia de entidades físicas en el sistema (como sensores) o son conjurados a partir de la experiencia y/o intuición del diseñador. Por el contrario, según la metodología propuesta por Parnas y basada en el POI los módulos se determinan como consecuencia de ocultar los ítem con alta probabilidad de cambio. Esto no es un detalle menor: cómo obtener los módulos del sistema es diseñar, por lo tanto un método de diseño que no indique una forma rigurosa para obtener lo módulos está lejos de ser un buen método de diseño.

Sin embargo el DBOI tiene limitaciones. En realidad, no es del todo correcto hablar de limitaciones del DBOI, mejor sería decir que, como lo hemos planteado, esta forma de diseño tiene algunos inconvenientes que pueden ser fácilmente evitados. Nos referimos, concretamente, a la falta de una forma simple de generar instancias de módulos; lo que es semejante a pensar los módulos como tipos.

Por ejemplo, si tenemos que diseñar un sistema que recibirá señales de 20 sensores iguales, tendríamos dos posibilidades:

- Incluir en todas las subrutinas de la interfaz del módulo correspondiente (Sensores) un parámetro para indicar el sensor sobre el cuál se debe aplicar la subrutina.
- Generar 20 módulos idénticos.

Ninguna de las dos alternativas tiene la elegancia y simplicidad que podría tenerse. Además, si deseáramos que otros módulos reciban un "sensor" como parámetro, no tendríamos forma de expresarlo. Por lo tanto, sería conveniente contar con una sintaxis y una semántica que nos permitieran tratar a cada módulo como un tipo del cual se pueden generar instancias y a estas usarlas como parámetros en subrutinas. En otras palabras, es conveniente acercarnos a las ideas más recientes sobre Tipos Abstractos de Datos (TAD), lenguajes de programación orientados a objetos, etc.

3 Diseño basado en tipos abstractos de datos (DTAD)

La única diferencia entre el DBOI y el Diseño basado en Tipos Abstractos de Datos (DTAD) es que en este último se asume que cada módulo del diseño es, genera o define un tipo. Al ser cada módulo un tipo entonces es posible definir variables y parámetros que tengan ese tipo. Por lo demás, el DTAD es idéntico al DBOI: su metodología, notación, conceptos, etc.

Definición 9. (Tipo). *En el contexto del DTAD, un tipo es un módulo.*

Concretamente, si A es un tipo de un DTAD, entonces $a : A$ significa que a es una instancia de A. Si a es una instancia de A y f es una subrutina en la interfaz de A, entonces *a.f(. . .)* es lo mismo que f*(a, . . .)*.

Además, si b es otra instancia de A (es decir tiene un nombre diferente al de a) entonces $a.f$ no tiene necesariamente el mismo valor o efecto que $b.f$. Esto se interpreta diciendo que cada instancia de A tiene su propio estado y que el efecto o resultado de cualquier subrutina en la interfaz de A puede depender del estado de la instancia sobre la cual se aplica.

4 Diseño orientado a objetos (DOO)

El problema de la duplicación del código de los clientes así como también otros problemas (menores) de diseño pueden resolverse aplicando el concepto de herencia desarrollado con el correr de los años por la comunidad de profesionales y científicos dedicados al diseño y programación orientados a objetos.

Definición 10. (Herencia (de interfaces)). *Sean A y B dos tipos de un DTAD. Decimos que B es un heredero de A si toda subrutina de la interfaz de A es una subrutina de la interfaz de B.*

Si B es un heredero de A, f es una subrutina en la interfaz de A y $b : B$ entonces $b.f$ está definido. Esto trae como corolario que si g es una subrutina cualquiera tal que uno de sus parámetros formales es de tipo A, entonces también aceptará un parámetro real de tipo B (porque se supone que externamente un tipo (módulo) queda definido por su interfaz, por lo que para un componente externo dos instancias con la misma interfaz deberían poder ser utilizadas de la misma forma aunque sean de tipos diferentes).

Definición 11. (Diseño Orientado a Objetos (DOO)). *Un DTAD en el cual se utiliza el concepto de herencia definido en la Definición 10, pasa a ser un Diseño Orientado a Objetos (DOO).*

Definición 12. (Objeto). *En el contexto del DOO, un objeto es una instancia de un tipo (módulo).*

Definición 13. (Supertipo y subtipo). *Si A y B son tipos de un DOO y B es un heredero de A decimos que B es un subtipo de A y que A es un supertipo de B. Si $A_1, ..., A_n$ son tipos de un DOO tales que A_i es un heredero de A_{i-1} para todo i, entonces decimos que A_i es un subtipo (supertipo) de A_i para todo $i > (<)j$.*

Definición 14. (Método). *En el contexto del DOO, un método es una subrutina en la interfaz de un tipo.*

La *herencia de clases* viola el POI porque permite a los herederos acceder a la implementación del padre. Por el contrario, la *herencia de interfaces* describe cuándo se puede usar un objeto de un tipo en lugar de un objeto de otro tipo.

Los herederos(de interfaz) pueden modificar la implementación de cualquier método de la interfaz.

En cualquier parte del programa donde se use algo del tipo A, puedo usar algo del tipo B sii B es heredero de A.

La herencia de interfaces es una relación estática entre módulos.

La herencia de interfaces resuelve el problema de duplicación de código cliente.

Principio de sustitución de Liskov Si S es un subtipo de T, entonces los objetos de tipo T pueden ser sustituidos por objetos de tipo S sin alterar ninguna de las propiedades importantes del programa.

¿Como usar la herencia en la metodología de Parnas?

1. Se considera cierto item de alta probabilidad de cambio.
2. Se define un tipo que lo oculta, pero que en general, no será implementado, solo provee una interfaz.
3. Para cada variante de item de cambio, se define un heredero que por lo general preserva la interfaz del padre.
4. Se implementan los herederos.
5. Los clientes de todos estos tipos, se programan en términos del supertipo.

Principio de Programación Orientada a Objetos (POO) Programe para una interfaz, no para una implementación. Es decir, no se deben declarar las variables con el tipo de los herederos sino con el tipo de los supertipos.

Herencia en 2MIL: **COMPLETAR**

El uso abusivo de la herencia tiende a generar malos diseños. En efecto, la herencia es un concepto estático porque las relaciones de herencia quedan congeladas en tiempo de compilación. No se pueden alterar las relaciones de herencia dinámicamente. Por lo tanto, un diseño fuertemente basado en la herencia tenderá a generar un sistema no muy flexible en tiempo de ejecución. Por otro lado, un uso exagerado de la herencia da lugar a jerarquías hereditarias de varios niveles lo que resulta difícil de comprender y mantener y, por lo general, implica que los módulos no ocultarán un único ítem con alta probabilidad de cambio sino varios. Nuevamente, si se utilizan variantes de la herencia de clases todo este panorama empeora porque, por ejemplo, un cambio en la implementación de un supertipo incide en la implementación de todos sus subtipos.

La herencia debe utilizarse en su justa medida, no es el concepto del DOO ni es la bala de plata del diseño. La herencia se complementa muy bien con la composición de objetos.

4.1 Composición de objetos

Hay composición de objetos cuando el estado o comportamiento de un objeto, de tipo T, dependa del estado o comportamiento de un objeto de otro tipo.

Pasar un objeto como parámetro a un método no necesariemente constituye composición, lo es solo si el objeto pasado se convierte en parte del estado del objeto que lo recibe.

Composición de objetos es una relación dinámica entre objetos. (Pueden cambiar en tiempo de ejecución)

Principio de Composición de objetos Favorecer la composición de objetos frente a la herencia (de clases).

Es decir que se debe privilegiar el uso de composición de objetos como mecanismo para compartir código por sobre la herencia de clases.

Las ventajas de la composición frente a la herencia de clases son:

1. Se logran diseños mucho más flexibles, sobre todo en tiempo de ejecución.
2. Se preserva el encapsulamiento de cada módulo, no se rompe el principio de ocultación de información.
3. Requiere interfaces cuidadosamente definidas lo que implica tomar el diseño en serio.
4. Las jerarquías de herencia tienden a ser pequeñas y manejables.

Muchas veces la composición se utiliza para redirigir peticiones de servicio hacia el objeto interior. (Ver ejemplo)

5 Tópicos complementarios

5.1 Efectos laterales

Se dice que un método o un objeto tiene o produce un efecto lateral cuando tiene una interacción observable con el mundo exterior. En este caso el mundo exterior es cualquier software no orientado a objetos. Es decir que un objeto, a través de uno o más de sus métodos, produce un efecto lateral cuando debe interactuar con un programa que no puede hacerlo por medio de objetos. Como este otro programa no reconoce objetos (y por consiguiente tampoco sus tipos), el programa orientado a objetos deberá convertir objetos en no-objetos (cosas no orientadas a objetos, entes no tipados, o con tipos fuera del sistema de tipos del programa y del lenguaje de programación) o viceversa.

Gran parte del diseño y del código de un programa orientado a objetos está relacionado de una u otra forma con efectos laterales. Es decir que gran parte del código no trabaja con objetos o debe hacer conversiones en ambos sentidos entre objetos y no-objetos. En consecuencia, los efectos laterales contaminan el diseño pues incluyen código que no puede respetar el sistema de tipos ya que el entorno con el cual interactúa tampoco lo hace. Algunos efectos laterales que aparecen en casi todos los programas útiles son:

- **Procesar la entrada del usuario**. En general el usuario se comunica con el programa por medio de cadenas de caracteres y eventos del mouse. Ninguna de estas cosas son objetos.
- **Emitir la salida para el usuario**. El usuario no puede ver objetos en la pantalla ni puede escuchar objetos a través de un parlante, por lo tanto un programa orientado a objetos deberá convertir los objetos en sonido o cadenas de caracteres o de enteros.
- **Almacenar o recuperar datos de almacenamiento secundario**. Ni los sistemas de archivos ni los motores de bases de datos son orientados a objetos (al menos los que se usan comúnmente). Ni un archivo ni una tabla son objetos ni tienen tipo (del sistema de tipos del programa orientado a objetos que los quiere utilizar). En consecuencia si un programa orientado a objetos debe persistir un objeto en un archivo o en una tabla, deberá convertirlo en no-objetos; y deberá hacer lo inverso cuando quiera recuperarlos.

Claramente, los efectos laterales son tan inevitables como contaminantes. La estrategia que debe seguirse en el **DOO** es: aislar las porciones de código que producen efectos laterales lo más posible y lo antes posible, en el caso de las entradas y lo más tarde posible en el caso de las salidas.

De hecho la metodología de Parnas predice este tipo de diseños: los efectos laterales son producto de representar cierto ente del mundo real con una cierta porción de código y con ciertas estructuras de datos. Esa representación es arbitraria y por ende es un ítem de alta probabilidad de cambio. En consecuencia debe ser aislado en un módulo con una interfaz abstracta insensible a los cambios anticipados. Una vez que el dato proveniente del exterior se ha tipado, es decir se ha convertido en un objeto (es decir, se ha anulado o resuelto el efecto lateral), entonces puede ser convenientemente manejado por el resto del sistema... hasta que debe ser expulsado al exterior.

5.2 Generadores y observadores

Un *generador* es un método que modifica el estado de las instancias del TAD sin retornar nada. Un *observador* es un método que retorna un valor de otro TAD sin modificar el estado de la instancia sobre la cual se invoca. Si un método no cae dentro de ninguna de estas categorías entonces está mal definido (al menos desde el punto de vista teórico).

Por otro lado se vió que la interfaz de un módulo debe estar compuesta por la mínima cantidad de métodos que permitan proveer todos los servicios requeridos. Si los métodos se dividen en generadores y observadores, entonces la interfaz de un TAD debe estar compuesta por:

- La cantidad mínima de generadores tales que se pueda poner a una instancia dada en cualquiera de los estados admitidos por la especificación.
- La cantidad mínima de observadores tales que permitan observar el comportamiento externo especificado para el TAD.

Según estas definiciones un generador no puede retornar un error.

5.3 Excepciones

Las excepciones no son un concepto perteneciente a la esfera del diseño por lo que solo mencionaremos un aspecto relacionado con el tema de la sección anterior. Muchos lenguajes de programación orientada a objetos introducen el mecanismo de las excepciones para comunicar errores durante la ejecución de métodos. Por otra parte, los valores de retorno de un método podrían usarse para comunicar errores. Surge entonces el dilema de cuál mecanismo usar para comunicar errores. Para dirimir este dilema es necesario tener en cuenta que cada método (tanto generadores como observadores) puede ser, matemáticamente hablando, una función total o una función parcial. Una función es parcial si no está definida para todo su dominio, caso contrario es total.

Por ejemplo, el método (usualmente llamado head()) que toma una lista de elementos y retorna el primero de ellos, es una función parcial pues si la lista está vacía no puede retornar nada. Este método puede definirse de tres formas diferentes:

- head(o Elemento):Bool

  En este caso el valor de retorno indica si hay error o no; si no lo hay el valor retornado por el parámetro es válido, y es inválido en cualquier otro caso.

- head():Elemento – lanzando una excepción.

  Si el método es invocado sobre una lista vacía lanza una excepción indicando esta situación.

- head():Elemento – sin lanzar excepción.

  Los invocantes del método son responsables de verificar, previo a la llamada, que esta se puede realizar de forma segura. Si lo hacen en un estado incorrecto, el TAD al cual pertenece head() no garantiza nada.

En los dos primeros casos, el método se vuelve una función total, es decir brinda un comportamiento aceptable para todo su dominio. En el tercero sigue siendo una función parcial. Este último caso es razonable únicamente cuando los invocantes van a ser programados por el mismo equipo de ingenieros, quienes se responsabilizan por verificar la precondición antes de cada llamada.

Las dos primeras posibilidades son razonables en cualquier caso pero la segunda es la más conveniente. En efecto, las excepciones fueron creadas para dotar de un comportamiento razonable a los métodos parciales por lo que la segunda opción es la correcta.

Por este motivo, los generadores solo devolverán excepciones en tanto que los observadores utilizarán los valores de retorno para devolver valores útiles y las excepciones para comunicar errores.

6 Límites del DOO

El DOO tiene varias deficiencias técnicas, metodológicas y expresivas que, mirando el desarrollo de un sistema globalmente, nos llevan a tomar el **DOO** como una forma de diseño de bajo nivel de abstracción.

El DOO permite conectar módulos únicamente por medio de llamada a procedimiento.

Las interfaces de los módulos de un DOO quedan congeladas en tiempo de compilación. Los módulos de un DOO no tienen ningún tipo de restricción estructural (cualquier módulo puede relacionarse con cualesquiera otros) es decir el DOO presenta un espacio de soluciones homogéneo e isotrópico. Todas estas características, llegado el momento de diseñar grandes o complejos sistemas de software, son desventajas que pueden hacer al fracazar el proyecto.

La comunidad del DOO logró dar un paso en la dirección correcta al introducir los patrones de diseño. Sin embargo, este salto en el nivel de abstracción y estructuración del espacio de soluciones, no es suficiente para diseñar grandes sistemas. Al mismo tiempo, no se puede lograr el nivel de abstracción y estructuración del espacio de soluciones adecuados sin salirse del DOO. En este sentido, se necesita echar mano de los conceptos de Arquitectura de Software y Estilos Arquitectónicos que veremos más adelante. Cuando terminemos de recorrer el camino que va desde el DBOI, pasando por el DOO y los patrones de diseño, hasta los estilos arquitectónicos, entenderemos que el DOO es, como dijimos, una forma de diseño de bajo nivel de abstracción pero que subyace sobre todas las formas de arquitectura y diseño. Es decir, los principios del DOO son aplicables a todos los sistemas pero en el nivel de abstracción correcto y en la etapa de desarrollo apropiada.

7 Documentación de diseño

Principio: Diseñar es documentar Un diseño sin documentación carece de utilidad práctica.

Un diseño no documentado carece de utilidad práctica porque la implementación debe basarse únicamente en la documentación de diseño. Sin documentación de diseño, los programadores no deberían iniciar la implementación

Como menciona la definición de diseño, este incluye la especificación funcional de cada uno de sus elementos. La forma de documentar la función asignada a cada módulo es dando una especificación formal de su comportamiento. Como especificaciones formales ya lo hemos estudiado extensivamente, no mencionaremos más en este documento este aspecto de la documentación de diseño.

7.1 Documentos

La documentación de diseño se compone de documentos cada uno de los cuales describe el diseño del sistema desde un punto de vista particular, y de documentos que relacionan uno o más documentos. Cuantos más documentos se generen, más completa estará la documentación. Sin embargo, para ciertos sistemas no es necesario describir muchos de ellos.

Documentos de módulos. Los elementos de estos documentos son módulos o unidades de implementación. Los módulos representan una forma basada en el código de considerar al sistema. Cada módulo tiene asignada y es responsable de llevar adelante una función.

Documentos de aspectos dinámicos. En estos documentos los elementos son componentes presentes en tiempo de ejecución y los conectores que permiten que los componentes interactúen entre sí.

Documentos con referencias externas. Estos documentos muestran la relación entre las partes en que fue descompuesto el sistema y elementos externos (tales como archivos, procesadores, personas, etc.)

Documentos de módulos. Especificación de Interfaces, Estructura de Módulos, Guía de Módulos, Estructura de Herencia y Estructura de Uso.

Documentos de aspectos dinámicos. Estructura de Procesos, Estructura de Objetos, Diagrama de Interacciones.

Documentos con referencias externas. Estructura Física o de Despliegue

En cuanto a la documentación que relaciona dos o más de los documentos descriptos más arriba sólo diremos que si el nombre de una entidad en un documento coincide con el nombre de una entidad en otro documento, entonces se trata de la misma entidad vista desde dos ángulos diferentes.

7.2 Estructura de Módulos

Agrupar módulos según algún criterio. A partir de esto se consiguen, módulos lógicos(o impropios).

Los otro tipo de módulos que vimos son módulos físicos(o propios).

Los segundos vienen a ser como una especie de directorio. Contienen otros módulos lógicos y/o módulos físicos.

Para esto tenemos la palabra reservada **comprises**, por ejemplo:

Module	**System**
comprises	Client
	Server

Carpeta raíz de un sistema *Cliente-Servidor*.

A su vez, *Client* y *Server* pueden ser tanto módulos lógicos o físicos.

El criterio debería ser una generalización del principio de ocultación de información.

Los módulos lógicos no tienen interfaz, pero pueden utilizar la cláusula **exports** de la siguiente manera:

Module	**System**
comprises	Client
	Server
exportsproc	Client
	f()

Esto significa que todas las subrutinas en la interfaz del módulo **Client** más la subrutina f() son las únicas subrutinas de los submódulos de **System** que pueden ser accedidas por módulos que no pertenezcan a **System**. Esto se usa para definir una *fachada*.

7.3 Guía de Módulos

La Guía de Módulos es un documento escrito en lenguaje natural que complementa la Especificación de Interfaces y la Estructura de Módulos. El objetivo de este documento es permitir que los diseñadores, programadores y mantenedores del sistema puedan identificar las partes del software que ellos necesitan entender sin tener que leer detalles irrelevantes sobre las otras partes.
Módulos lógicos. Descripción del criterio por el cual el módulo contiene a sus "hijos". Muy breve descripción de los módulos existentes y de su función (uno o dos renglones por hijo).
Módulos físicos. El texto se divide en dos secciones: Función y Secreto.

- **Función.** En este apartado se da una especificación funcional del módulo lo más detallada posible. En principio se debe dar una especificación funcional de cada una de las subrutinas en la interfaz del módulo. Esta especificación funcional debe incluir, como mínimo, el significado de cada parámetro y una explicación de cada una de las salidas; en particular deben explicarse cada uno de los mensajes de error producidos (un mensaje de error no es únicamente algo que se imprime en la pantalla, sino los errores en general que emite la subrutina incluyendo valores de retorno, excepciones, etc.). En caso de que exista una especificación formal se debe incluir una referencia a aquella.

- **Secreto.** En la segunda se explica la decisión de diseño que se decidió ocultar y la razón que llevó a ello; debe haber una referencia al documento Análisis de Cambio. Si un módulo físico es heredero de otro, su descripción en la guía se hace al mismo nivel que la de su padre. Además de la información mencionada debe agregarse el motivo por el cual el módulo es un heredero y cuáles métodos se reimplementan.

7.4 Estructura de Uso

La Estructura de Uso fue propuesta originalmente por Parnas como medio para poder calcular los subconjuntos útiles y los incrementos mínimos de un sistema, y así habilitar la posibilidad de entregas

incrementales.

Definición 15. Un *subconjunto útil de un sistema* es un conjunto mínimo de subrutinas que proveen una funcionalidad útil para el destinatario del sistema. Es mínimo porque si se quita una de las subrutinas, la funcionalidad no puede realizarse. En la actualidad a los subconjuntos útiles se los denomina versiones del sistema.

Definición 16. Un *incremento mínimo* con respecto a un subconjunto útil, es un conjunto mínimo de subrutinas tal que si es agregado al subconjunto útil genera un nuevo subconjunto útil.

Una correcta ingeniería de subconjuntos útiles e incrementos mínimos permite realizar entregas incrementales al cliente, definir productos con menos o más funcionalidades orientados a diferentes clientes o hardware, etc.

La Estructura de Uso es la representación (usualmente gráfica o tabular) de la relación binaria entre unidades ejecutables $REQUIERE\ EJECUCION\ CORRECTA\ DE$ o REC. Las unidades ejecutables son subrutinas o programas (es decir cualquier unidad de código cuya ejecución pueda ser invocada desde algún punto de un programa).

Definición 17. Si f y g son dos unidades ejecutables, $(f, g) \in REC$ sí y sólo sí f necesita de una implementación correcta de g para poder cumplir con su especificación. Una unidad ejecutable f necesita de una implementación correcta de otra unidad ejecutable, sólo si la especificación de f así lo indica.

Es importante resaltar que la relación REC está asociada con la especificación de las unidades ejecutables. Esto significa que si la especificación de una subrutina cambia (por ejemplo porque el cliente quiere una versión inicial diferente de la final) la Estructura de Uso probablemente también lo hará. Sin embargo, aun en estos casos, esta vista constituye una ayuda invaluable pues, precisamente, nos muestra cuáles especificaciones dependen de otras.

Cabe preguntarse cuáles son las condiciones que deben darse para que $(f, g) \in REC$:

- f es sustancialmente más simple al usar g.

  Si f usa a g significa que el código de g no estará dentro de f y por lo tanto f debería ser más simple de comprender.

- g no es sustancialmente más compleja al no permitírsele usar f.

  Como veremos más abajo, no es conveniente que se generen ciclos en la estructura de uso. Por lo tanto, si $(f, g) \in REC$ entonces no convendrá que $(g, f) \in REC$. Entonces puede ocurrir que g termine siendo demasiado compleja porque no puede usar a f.

- Existe un subconjunto útil que contiene a g pero no contiene a f.

- No existe un subconjunto útil razonable que contiene a f pero no contiene a g.

$(f, g) \in REC$ puede darse sin que f llame a g. Mas aún, si f llama a g no necesariamente $(f, g) \in REC$.

7.5 Estructura de Procesos

La Estructura de Procesos permite tener una idea de cómo se comportará el sistema en tiempo de ejecución. Es una visión de alto nivel de abstracción puesto que no permite ver dentro de cada proceso. Por ejemplo, esta vista no permite saber cuántos o cuándo serán creados los objetos de cada tipo. Es una de las estructuras propuestas originalmente por Parnas. Aquí, consideraremos procesos con un único hilo de control por lo que procesos con múltiples hilos de control deben ser considerados como varios procesos.

Esta estructura o vista es la representación (usualmente gráfica) de la relación binaria entre procesos $SINCRONIZA\ CON$ o $SINC$.

Definición 18. Un proceso P sincroniza con otro proceso Q sí y sólo sí P emite un evento que Q espera. En este caso decimos que $(P, Q) \in SINC$. Notar que la definición de sincronización es idéntica a la utilizada en CSP. Si dos procesos, P y Q, son tales que $(P, Q) \notin SINC$ entonces decimos que son independientes y por lo tanto pueden ejecutar en completo paralelismo.

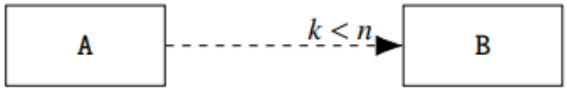
7.6 Estructura de Objetos

Esta vista aplica sólo a los DTAD y DOO pues relaciona las instancias (objetos) de los diversos tipos existentes en el sistema. Como ya observamos tanto en la Especificación de Interfaces como en la Estructura de Módulos, no hay posibilidad de conocer cuántas instancias de cada módulo se crearán (lo que muchas veces se hace recién en tiempo de ejecución) y qué relación hay entre ellas. La Estructura de Objetos viene a completar esta falta.

Esta vista se relaciona con la Estructura de Procesos pues los objetos se crean dentro de un proceso por lo cual es importante dejar registrado en relación a cuál de los procesos del sistema pertenecen los objetos que se mencionan. Dividimos la Estructura de Objetos en dos documentos.

- **Creación de objetos.** En este documento se describe cuáles módulos crean objetos de otros tipos.

donde se indica que el módulo A crea uno o más objetos de tipo B. Si conocemos la cantidad de objetos que se crean de forma más precisa podemos anotar la flecha con el número o expresión correspondiente.



Como los objetos pueden ser creados y destruidos durante la ejecución del sistema, hay que dejar en claro qué se entiende por la cantidad de objetos que se crearán: ¿coexistirán simultáneamente? ¿Es el total que se creará durante la vida del proceso? No importa cuál sea la semántica que se le dé, lo importante es que se documente y que la información consignada sea consistente con esa semántica.

- **Relaciones entre objetos.** En un segundo documento se describen las relaciones entre los objetos de los distintos tipos. No siempre es posible hacerlo y no siempre es necesario hacerlo para todos los objetos del sistema. Generalmente solo se documentan las relaciones más complejas o que son más difíciles de comprender. Las relaciones pueden ser $COMPUESTO\ CON$, que indica alguna forma de composición, y $ENLAZA\ CON$ que engloba a las relaciones por las cuales unos objetos utilizan los servicios de otros.

7.7 Estructura de Herencia

Esta estructura muestra las relaciones de herencia que hay entre los diferentes tipos. Usualmente sólo hay unas pocas "líneas hereditarias" que involucran a unos pocos tipos por lo que suele ser adecuado una representación gráfica en forma de árbol para cada familia. Si se describieron las interfaces con 2MIL esta estructura se puede obtener por análisis sintáctico.

Los módulos que solo ofrecen una interfaz (es decir que no se implementan) no deben importar ningún recurso.

7.8 Estructura de Física

Esta vista permite relacionar elementos de software con plataformas de ejecución tales como procesadores, discos, redes, etc. Usualmente los elementos de software son procesos de la Estructura de Procesos. La relación más común que se utiliza para documentar esta vista es $ALOJADO\ EN$, la cual relaciona un elemento de software (generalmente un proceso) con un elemento de hardware (generalmente un procesador o computadora).

7.9 Líneas y Cajas

Muchas de las estructuras que hemos mencionado se pueden documentar gráficamente. Estos gráficos toman la forma de un grafo donde los nodos se dibujan como cajas que pueden ser procesos, unidades ejecutables, módulos, etc.

Si bien un gráfico suele ser muy bueno como vehículo para comunicar el diseño de un sistema, tienen problemas de expresividad y de semántica.

Por lo tanto, todo gráfico debe ir acompañado de una referencia que explique qué son las cajas y qué significan las flechas. Esta explicación debe apuntar a dar un significado claro, unívoco y preciso de esos elementos,

El problema de la falta de significado de gráficos suele ser recurrente en el uso práctico (no tanto en la teoría) de UML.

7.10 Nota sobre la protocolización de la documentación de diseño

Todo lo que hemos mencionado en esta sección sobre documentación de diseño refiere únicamente al contenido técnico de cada documento. Sin embargo, cada documento debe estar precedido por datos protocolares tales como: nombre y apellido de quién es el responsable del documento, propósito, número de versión, fecha de la última modificación, etc.

8 Patrones de diseño

Son soluciones estándar y probadas de problemas de diseño que aparecen frecuentemente.
En el parcial entran los patrones:

- *Abstract Factory*
- *Bridge*
- *Composite*
- *Decorator*
- *Command*
- *Iterator*
- *Strategy*
- *Visitor*

Clasificación

1. Patrones de creación: *Abstract Factory*, *Builder*, *Factory Method*, *Prototype* y *Singleton*.
2. Estructural de objetos: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* y *Proxy*
3. Comportamiento de objetos: *Chain of Responsability*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template Method* y *Visitor*.

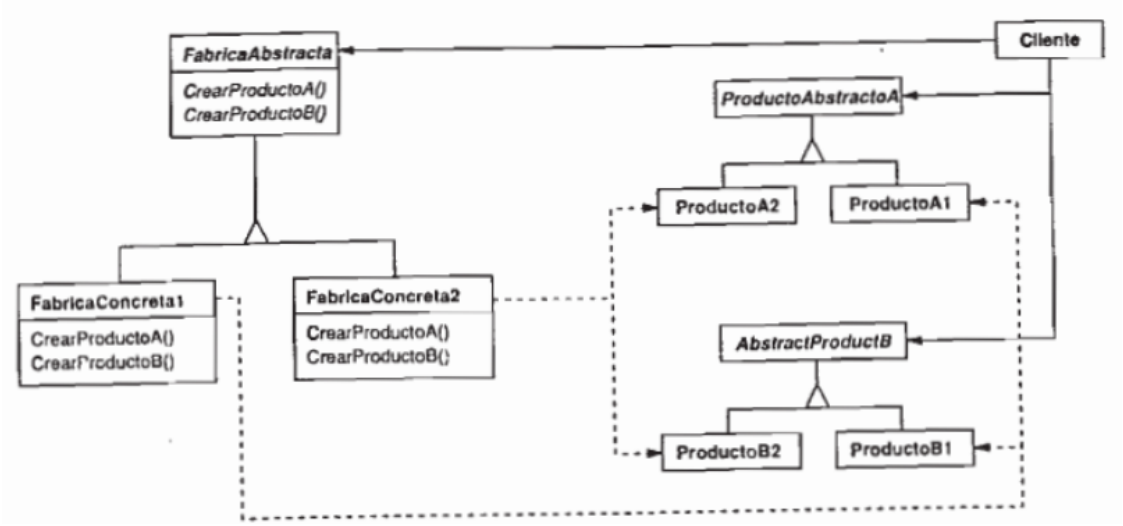
8.1 Patrones de Creación

8.1.1 Abstract Factory

También llamado *Kit*.
Este patrón proporciona una inerfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
Aplicabilidad. Use el patrón cuando:

1. Un sistema debe ser independiente de cómo se crean, componen y representan sus productos.
2. Un sistema debe ser configurado con una familia de productos de entre varias.
3. Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplis esta restricción.
4. Quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones.

Estructura.



Participantes.

- **FabricaAbstracta**
  - Declara una interfaz para operaciones que crean objetos producto abstractos.
- **FabricaConcreta**
  - Implementa las operaciones para crear objetos producto concretos.
- **ProductoAbstracto**
  - Declara una interfaz para un tipo de objeto producto.
- **ProductoConcreto**
  - Define un objeto producto para que sea creado por la fábrica correspondiente.
  - Implementa la interfaz ProductoAbstracto.
- **Cliente**
  - Sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.

Colaboraciones.

Normalmente sólo se crea una única instancia de una clase FabricaConcreta en tiempo de ejecución. Esta fábrica crea objetos producto que tienen una determinada implementación.

FabricaAbstracta delega la creación de objetos producto en su subclase FabricaConcreta.

Consecuencias. El patrón Abstract Factory presenta las siguientes ventajas e inconvenientes:

1. *Aísla las clases concretas.* El patrón Abstract Factory ayuda a controlar las clases de objetos que crea una aplicación. Como una fábrica encapsula la responsabilidad y el proceso de creación de objetos producto, aísla a los clientes de las clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas. Los nombres de las clases producto quedan aisladas en la implementación de la fábrica concreta; no aparecen en el código cliente.

2. *Facilita el intercambio de familias de productos.* La clase de una fábrica concrete sólo aparece una vez en una aplicación. Esto facilita cambiar la fábrica concreta que usa una aplicación. Como una fábria abstrata rea una familia completa de productos, toda la familia de productos cambia de una vez.

3. *Promueve la consistencia entre productos.* Cuando se diseñan objetos producto en una familia para trabajar juntos, es importante que una aplicación use objetos de una sola familia a la vez. FabricaAbstracta facilita que se cumpla esta restricción.

4. *Es difícil dar cabida a nuevos tipos de productos.* Ampliar las fábricas abstractas para producir nuevos tipos de productos no es fácil. Esto se debe a que la interfaz FabricaAbstracta fija el conjunto de productos que se pueden crear. Permitir nuevos tipos de productos requiere ampliar la interfaz de la fábrica, lo que a su vez implica cambiar la clase FabricaAbstracta y todas sus subclases.

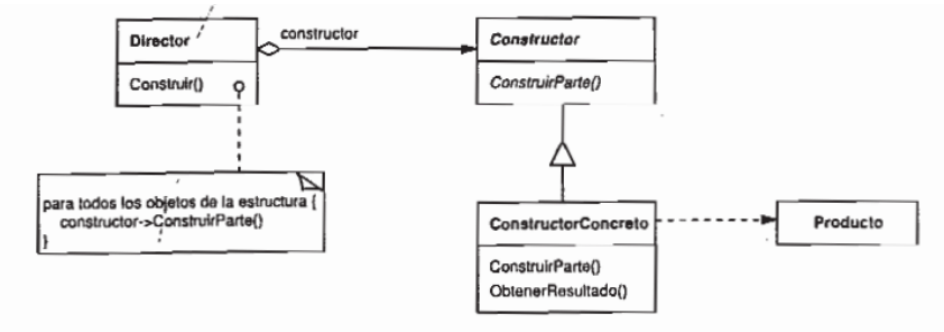
8.1.2 Builder

Separa la construcción de un objeto completo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Aplicabilidad. Use el patrón cuando:

1. El algoritmo para crear un objeto complejo debiera ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
2. El proceso de construcción debe permitir diferentes representaciones del objeto que está siento construido.

Estructura.



Participantes.

- **Constructor**
  - Especifica una interfaz abstracta para crear las partes de un objeto Producto.
- **ConstructorConcreto**
  - Implementa la interfaz Constructor para construir y ensamblar las partes del producto.
  - Define la representación a crear.
  - Proporciona una interfaz para devolver el producto.
- **Director**
  - Construye un objeto usando la interfaz Constructor.

Colaboraciones.

El cliente crea el objeto Director y lo configura con el objeto Constructor deseado.
El Directos notifica al constructor cada vez que hay que construir una parte de un producto.
El Constructor maneja las peticiones del director y las añade al producto.
El cliente obtiene el producto del constructor.

Consecuencias.

- *Permite variar la representación interna de un producto.* El objeto Constructor proporciona al director una interfaz abstracta para construir el producto. La interfaz permite que el constructor oculte la representación y la estructura interna del producto. También oculta el modo en que éste es ensamblado. Dado que el producto se contruye a través de una interfaz abstracta, todo lo que hay que hacer para cambiar la representación interna del producto es definir un nuevo tipo de constructor.

- *Aísla el código de contrucción y representación.* Este patrón aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos. Los clientes no necesitan saber nada de las clases que definen la estructura interna del producto. Cada ConstructorConcreto contiene todo el código para crear y ensamblar un determinado tipo de producto. El código sólo se escribe una vez; después, los diferentes Directores pueden reutilizarlo para construir variante de Producto a partir del mismo conjunto de partes.

- *Proporciona un control más fino sobre el proceso de construcción.* El patrón construye el producto paso a paso, bajo el control del director. El director sólo obtiene el producto una vez este está terminado. Por lo tanto, la interfa Constructor refleja el proceso de construcción del producto más que otros patrones de creación. Esto da un control más fino sobre el proceso de construcción y, por tanto, sobre la estructura interna del producto resultante.

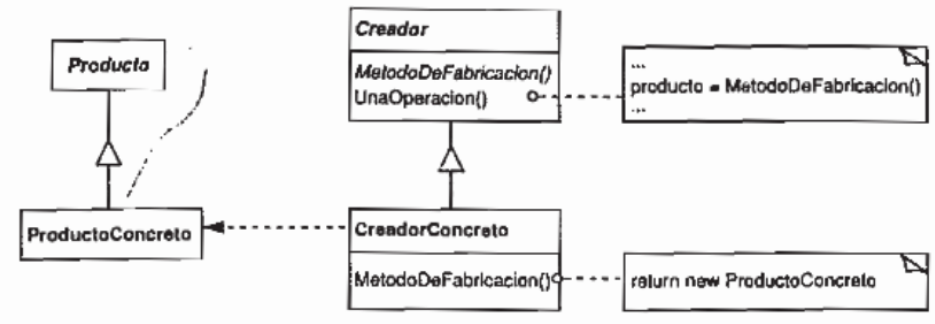
8.1.3 Factory Method

También conocido como *Virtual Constructor*.
Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Aplicabilidad. Use el patrón cuando:

1. Una clase no puede prever la clase de objetos que debe crear.
2. Una clase quiere que saen sus subclases quienes especifiquen los objetos que ésta crea.
3. Las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar qué subclase auxiliar concreta es en la que se delega.

Estructura.



Participantes.

- **Producto**
  - Define la interfaz de los objetos que crea el método de fabricación.
- **ProductoConcreto**
  - Implementa la interfaz Producto.
- **Creador**
  - Declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConcreto.
  - Puede llamar al método de fabricación para crear un objeto Producto.
- **Creador**
  - Redefine el método de fabricación para devolver una instancia de un ProductoConcreto.

Colaboraciones. El creador se apoya en sus subclases para definir el método de fabricación de manera que éste devuelva una instancia del ProductoConcreto apropiado.

Consecuencias.

- Los métodos de fabricación eliminan la necesidad de ligar clases específicas de la aplicación a nuestro código.
- El código no solo trata con la interfaz Producto, funciona con cualquier clsae ProductoConcreto.
- Los clientes pueden tener que heredar de la clase Creador simplemente para crear un determinado objeto ProductoConcreto.
- *Proporciona enganches para las subclases.* Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente. El Factory Method les da a las subclases un punto de enganche para proveer una versión extendida de un objeto.
- *Conecta jerarquías de clases paralelas.* Los clientes pueden encontrar útiles los métodos de fabricación, especialemente en el caso de jerarquías de clases paralelas. Dichas jerarquías de clases paralelas se producen cuando una clase delega alguna de sus responsabilidades a una clase separada.

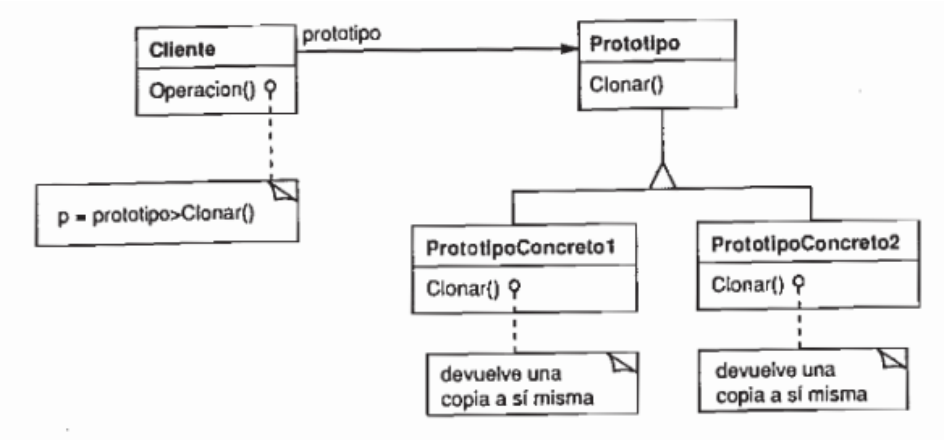
8.1.4 Prototype

Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.

Aplicabilidad. Use el patrón cuando un sistema deba ser independiente de cómo se crean, se componen y se representan sus productos y;

1. Cuando las clases a instanciar sean especificadas en tiempo de ejecución.
2. Para evitar construir una jerarquía de clases de fábricas paralela a la jerarquía de clases de los productos.
3. Cuando las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes. Puede ser más adecuado tener un número equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de la clase cada vez con el estado apropiado.

Estructura.



Participantes.

- **Prototipo**
  - Declara una interfaz para clonarse.
- **PrototipoConcreto**

  - Implementa una operación para clonarse.

- **Cliente**
  - Crea un nuevo objeto pidiéndole a un prototipo que se clone.

Colaboraciones. Un cliente le pide a un prototipo que se clone.
Consecuencias.

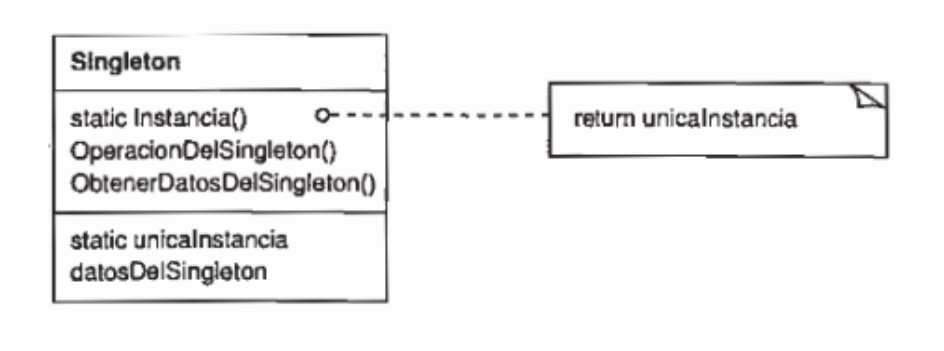
- *Añadir y eliminar productos en tiempo de ejecución.* Permite incorporar a un sistema una nueva clase concreta de producto simplemente registrado una instancia prototípica con el cliente.
- *Especificar nuevos objetos modificando valores.* Podemos definir nuevos tipos de objetos creando instancias de clases existentes y registrando estas como prototipos de los objetos clientes. Un cliente puede exhibir comportamiento nuevo delegando responsabilidad en su prototipo.
- *Especificar nuevos objetos variando la estructura.* Simplemente añadimos un subcircuito como prototipo a la paleta de circuitos disponibles. Siempre y cuando el objeto circuito compuesto implemente **Clonar** como una copia profunda, los circuitos con varias estructuras también pueden ser prototipos.
- *Reduce la herencia.*
- *Configurar dinámicamente una aplicación con clases.*

8.1.5 Singleton

Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.
Aplicabilidad. Use el patrón cuando:

1. Deba haber exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
2. La única instancia debería ser extensible mediente herencia, y los cliente deberían ser capaces de usar una instancia extendida sin modificar su código.

Estructura.



Participantes.

- **Singleton.**
  - Define una operación Instancia que permite que los clientes accedan a su única instancia. Instancia es una operación de clase.
  - Puede ser responsable de crear su única instancia.

Colaboraciones. Los clientes acceden a la instancia de un Singleton eclusivamente a través de la operación Instancia de éste.
Consecuencias.

- *Acceso controlado a la única instancia.* Puesto que la clase Singleton encapsula única instancia, puede tener un control estricto sobre cómo y cuándo acceden a ella los clientes.
- *Espacio de nombres reducido.* El patrón Singleton es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenen las instancias.
- *Permite el refinamiento de operaciones y la representación.* Se puede crear una subclase de la clase Singleton, y es fácil configurar una aplicación con una instancia de la clase necesaria en tiempo de ejecución.
- *Permite un número variable de instancias.* El patrón hace que sea fácil cambiar de opinión y permitir más de una instancia de la clase Singleton. Además, podemos usar el mismo enfoque para controlar el número de instancias que usa la aplicación. Sólo se necesitaría cambiar la operación que otorga acceso a la instancia Singleton.
- *Más flexible que la operaciones de clase.* Otra forma de empaquetar la funcionalidad de un Singleton es usar operaciones de clase.

8.2 Patrones Estructurales

8.2.1 Adapter

8.2.2 Bridge

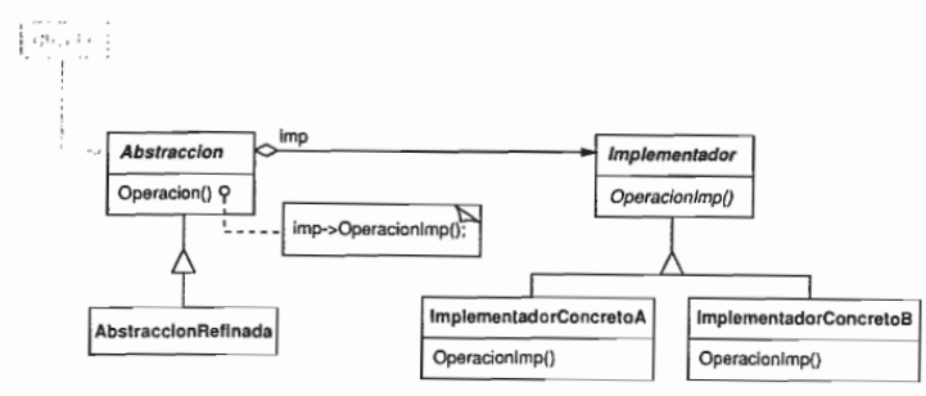
También conocido como *Handle/Body.*

Este patroón desacopla una astracción de su implementación, de modo que ambas puedan variar de forma independiente.

Aplicabilidad. Use el patrón cuando:

1. Quiera evitar un enlace permanente entre una abstracción y su implementación.
2. Tanto las abstracciones como sus implementaciones deberían ser extensibles mediante sublases.
3. Los cambios en la implementación de una abstracción no deberían tener impactos en los clientes; es decir, su código no tendría que ser recompilado.
4. Quiera compartir una implementación entre varios objetos y este hecho deba permanecer oculto al cliente.

Estructura.



Participantes.

- **Abstraccion**
  - Define la interfaz de la abstracción.
  - Mantiene una referencia a un objeto de tipo *Implementador*.

- **AbstraccionRefinada**
  - Extiende la interfaz definida por *Abstraccion*.
- **Implementador**
  - Define la interfaz de las clases de implementación. Esta interfaz no tiene por qué corresponderse exactamente con la de *Abstraccion*; de hecho, ambas interfaces pueden ser muy distintas. Normalmente la interfaz *Implementador* solo ofrece operaciones primitivas, y *Abstraccion* operaciones más complejas a partir de estas operaciones primitivas.
- **ImplementadorConcreto**
  - Implementa la interfaz *Implementador* y define su implementación concreta.

Colaboraciones. *Abstraccion* redirige las peticiones del cliente a su objeto *Implementador*.

Consecuencias. El patrón tiene las siguientes consecuencias:

1. *Desacopla la interfaz y la implementación.* No une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse en tiempo de ejecución. Incluso es posible que un objeto cambie su implementación en tiempo de ejecución.

   Desacoplar *Abstraccion* e *Implementador* también elimina de la implementación dependencias en tiempo de compilación. Ahora, cambiar una clase ya no requiere recompilar la clase *Abstraccion* y sus clientes.

   Este desacoplamiento también potencia una división en capas que puede dar lugar a sistemas mejor estructurados.

2. *Mejora la extensibilidad.* Podemos extender las jerarquías de *Abstraccion* y de *Implementador* de forma independiente.

3. *Oculta detalles de implementación a los clientes.* Podemos aislar a los clientes de los detalles de implementación, como el compotamiento de objetos implementadores y el correspondiente mecanismo de conteo de referencias (si es que hay alguno).

8.2.3 Composite

Este patrón compone objetos en estrctura de árbol para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

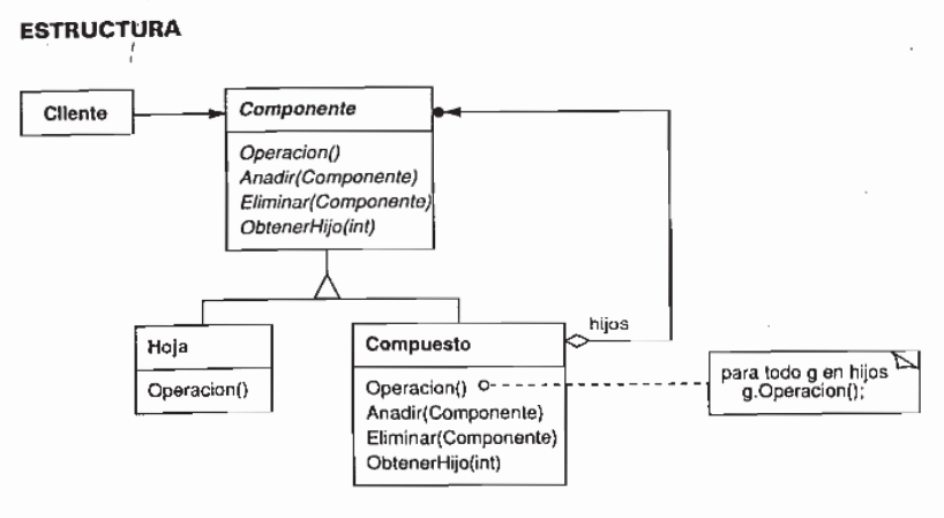
La clave del patrón *Composite* es un módulo abstracto que representa tanto a a primitivas como a sus contenedores.

Aplicabilidad. Este patrón se aplica cuando:

1. Se quiera representar jerarquías de objetos parte-todo.

2. Los clientes puedan obviar las diferencias entre grupos de objetos y los objetos individuales.

Estructura.



Se puede tener más de un tipo de hoja/compuesto.

Participantes.

- **Componente**
  - Declara la interfaz de los objetos de la composición.
  - Implementa el comportamiento que es común a todas las clases.
  - Declara una interfaz para acceder a sus componentes hijos y gestionarlos.
  - (Opcional) Declara una interfaz para acceder al padre de un componente de la estructura recursiva y, si es necesario, la implementa.
- **Hoja**
  - Representa una hoja en la composición. No tiene hijos.
  - Define el comportamiento de los objetos primitivos de la composición.
- **Compuesto**
  - Define el comportamiento de los objetos compuestos.
  - Almacena componentes hijos.
  - Implementa las operaciones de la interfaz *Componente* relacionadas con los hijos.
- **Cliente**
  - Manipula los objetos de la composición a partir de la interfaz **Componente**.

Colaboraciones. Los clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

Consecuencias. El patrón Composite:

1. *Define jerarquías formadas por objetos primitivos y compuestos.* Los objetos primitivos pueden componerse en otros objetos más complejos, que a su vez, pueden ser compuestos, y así de manera recurrente.
2. *Simplifica el cliente.* Los clientes pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales. Los clientes normalmente no conocen (y no les debería importar) si están tratando con una hoja o con un componente compuesto. Esto simplifica el código del cliente, puesto que evita tener que escribir funciones con instrucciones **if** anidadas en las clases que definen la composición.

3. *Facilita añadir nuevos tipos de componentes.*

4. *Puede hacer que un diseño sea demasiado general.* La desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto.

8.2.4 Decorator

También llamado *Wrapper*.
Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

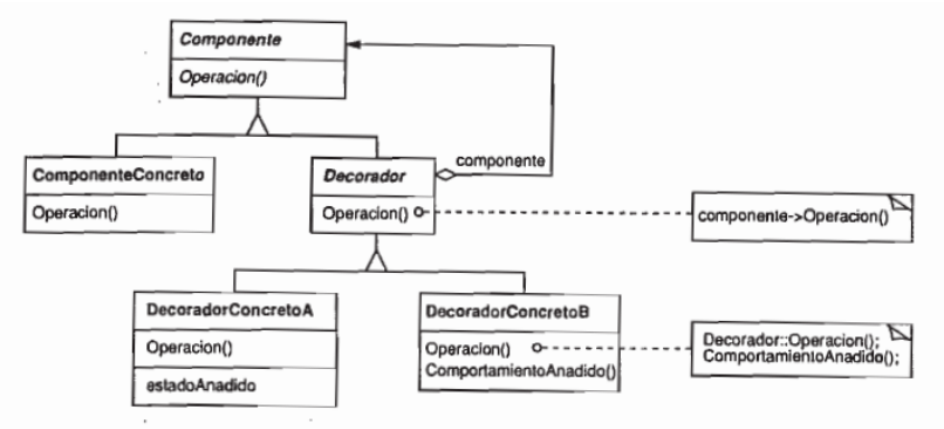
Aplicabilidad. Use el patrón cuando:

1. Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar otros objetos.

2. Para responsabilidades que pueden ser retiradas.

3. Cuando la extensión mediante herencia no es viable. A veces es posible tener un gran número de extensiones independientes, produciendose una explosión de subclases para permitir todas las combinaciones. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada.

Estructura.



Participantes.

- **Componente**
  - Define la interfaz para objetos a los que se le puede añadir responsabilidades dinámicamente.
- **ComponenteConcreto**
  - Define un objeto al que se le puede añadir responsabilidades adicionales.
- **Decorador**
  - Mantiene una referencia a un objeto **Componente** y define una interfaz que se ajusta a la interfaz de dicho componente.
- **DecoradorConcreto**
  - Añade responsabilidades al componente.

Colaboraciones. El Decorador redirige peticiones a su objeto Componente. Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.
Consecuencias.

1. *Más flexibilidad que la herencia estática.* Proporciona una manera más flexible de añadir responsabilidades a los objetos que la que podía obtenerse a través de la herencia (múltiple) estática. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas.

2. *Evita clases cargadas de funciones en la parte de arriba de la jerarquía.* Ofrece un enfoque para añadir responsabilidades que consiste en pagar sólo por aquello que se necesita. En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, podemos definir primero una clase simple y añadir lkuego funcionalidad incrementalmente con objetos Decorador. La funcionalidad se obtiene componiendo partes simples.

3. *Un decorador y su componente no son idénticos.* Un decorador se comporta como un revestimiento transparente. Pero desde el punto de viste de la identidad de un objeto, un componente decorado no es idéntico al componente en sí. Por tanto, no deberíamos apoyarnos en la identidad de objetos cuando estamos usando decoradores.

4. *Muchos objetos pequeños.* Un diseño que usa el patrón Decorator suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos. Los objetos sólo se diferencian en la forma en que están interconectados, y no en su clase o en el valor de sus variables.

8.2.5 Facade

8.2.6 Flyweight

8.2.7 Proxy

8.3 Patrones de Comportamiento

8.3.1 Chain of Responsability

8.3.2 Command

También conocido como *Action* y *Transaction*.

Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de peticiones, y poder deshacer las operaciones.

Aplicabilidad. Use el patrón cuando:

1. Permiten implementar el mecanismo de *Callback*.

2. Parametrizar objetos con una acción a realizar.

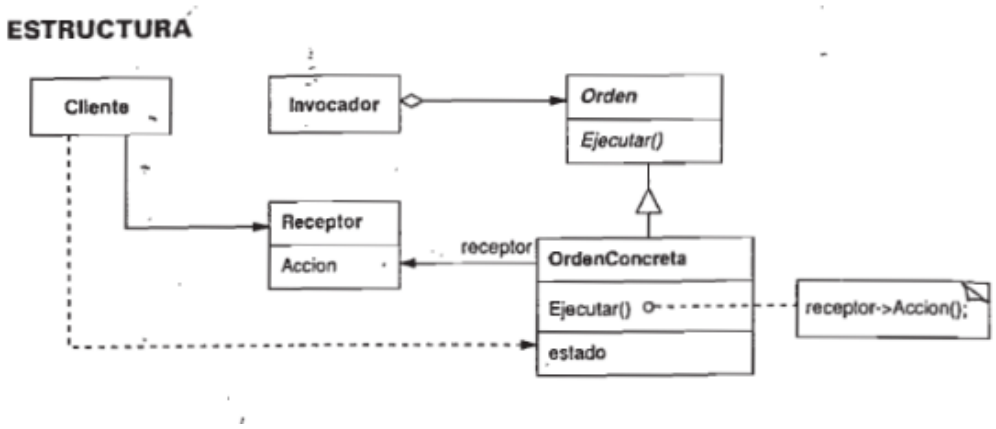
3. Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo.

4. Permitir deshacer.

5. Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída de un sistema.

6. Estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas. DIcha estructura es común en los sistemas de información que permiten transacciones.

Estructura.



Participantes.

- **Orden**
  - declara una interfaz para ejecutar una operación.
- **OrdenConcreta**
  - Define un enlace entre un objeto Receptor y una acción.
  - Implementa Ejecutar invocando la correspondiente operación u operaciones del Receptor.
- **Cliente**
  - Le pide a la orden que ejecute la petición.
- **Receptor**
  - Sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como Receptor.

Colaboraciones.

- El cliente crea un objeto OrderConcreta y especifica su receptor.
- Un objeto Invocador almacena el objeto OrdenConcreta.
- El invocador envía una petición llamando a Ejecutar sobre la orden. Cuando las órdenes se pueden deshacer, OrdenConcreta guarda el estado para deshacer la orden ante de llamar a Ejecutar.
- El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.

Consecuencias.

- Orden desacopla el objeto que invoca la operación de aquel que sabe cómo realizarla.
- Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto.
- Se pueden ensamblar órdenes en una orden compuesta.
- Es fácil añadir nuevas órdenes, ya que no hay que cambiar las clases existentes.

8.3.3 Interpreter

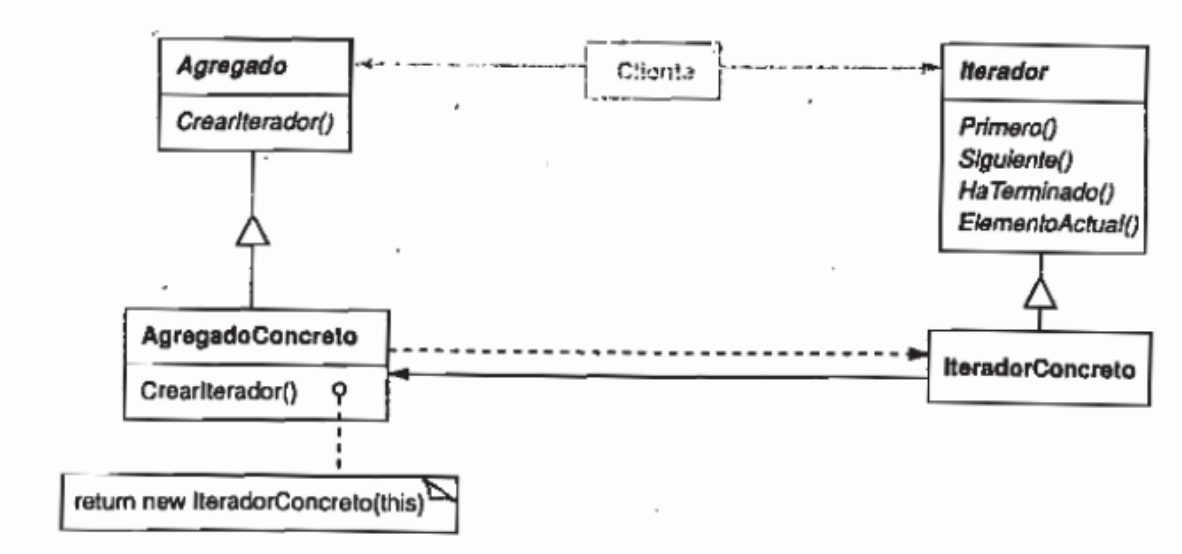
8.3.4 Iterator

También llamado *Cursor*.
Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

Aplicabilidad. Use el patrón cuando:

1. Para acceder al contenido de un objeto agregado sin exponer su representación interna.
2. Para permitir varios recorridos sobre objetos agregados.
3. Para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para permitir la iteración polimórfica).

Estructura.



Participantes.

- **Iterador**
  - Define una interfaz para recorrer los elementos y acceder a ellos.
- **IteradorConcreto**
  - Implementa la interfaz Iterador
  - Mantiene la posición actual en el recorrido agregado.
- **Agregado**
  - Define una interfaz para crear un objeto Iterador.
- **AgregadoConcreto**
  - Implementa la interfaz de creación de Iterador para devolver una instancia del IteradorConcreto apropiado.

Colaboraciones.
Un IteradorConcreto sabe cuál es el objeto actual del agregado y puede calcular el objeto siguiente en el recorrido.
Consecuencias.

- *Permite variaciones en el recorrido de un agregado.* Los agregados complejos pueden recorrerse de muchas formas.
- *Los iteradores simplifican la interfaz Agregado.* La interfaz de reocrrido de Iterador elimina la necesidad de una interfaz parecida en Agregado, simplificando así la interfaz del agregado.
- *Se puede hacer más de un recorrido a la vez sobre un agregado.*

8.3.5 Mediator

8.3.6 Memento

8.3.7 Observer

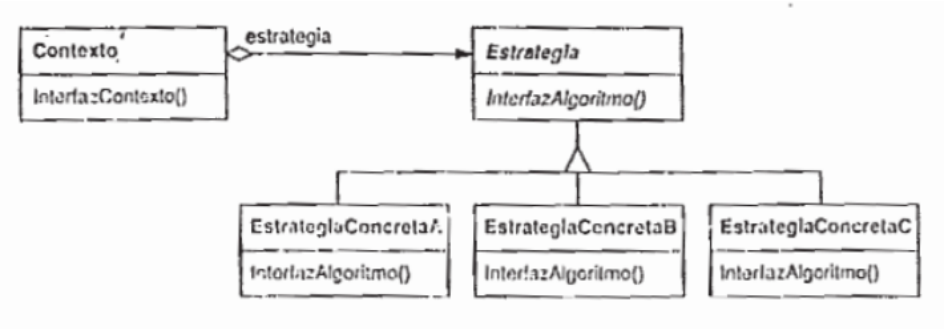
8.3.8 State

8.3.9 Strategy

También llamado *Policy*.
Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables.
Aplicabilidad. Este patrón se aplica cuando:

1. Muchas clases relacionadas difieren solo en su comportamiento.
2. Se necesitan distintas variantes de un algoritmo.
3. Un algoritmo usa datos que los clientes no deberían conocer.
4. Una clase define muchos comportamientos que se implementan con múltiples sentencias condicionales.

Estructura.



Participantes.

- **Estrategia**
  - Declara una interfaz común a todos los algoritmos permitidos.
  - El cliente usa esta interfaz para llamar a algún algortitmo definido en una estrategia.
- **EstrategiaConcreta**
  - Define una estrategia (algoritmo) concreta.
- **Contexto**
  - Se configura con un objeto **EstrategiaConcreta**.
  - Mantiene una referencia a un objeto **Estrategia**.
  - Puede definir una interfaz que permita a la Estrategia acceder a sus datos.

Colaboraciones. Estrategia y Contexto interactúan para implementar el algoritmo elegido. Un contexto puede pasar a la estrategia todos los datos requeridos por el algoritmo cada vez que se llama a éste. Otra alternativa es que el contexto se pase a sí mismo como argumento de las operaciones de Estrategia. Eso permite a la estrategia hacer llamadas al contexto cuando sea necesario.

Un contexto redirige peticiones de los clientes a su estrategia. Los clientes normalmente crean un objeto EstrategiaConcreta, el cual pasan al contexto; por tanto, los clientes interactúan exclusivamente con el contexto. Suele haber una familia de clases EstrategiaConcreta a elegir por el cliente.

Consecuencias. Este patrón presenta las siguientes ventajas e inconvenientes:

1. *Familias de algoritmos relacionados.* Las jerarquías de clases Estrategia definen una familia de algoritmos para ser reutilizados por los contestos.

2. *Una alternativa a la herencia.* La herencia ofrece otra forma de permitir una variedad de algoritmos. Se puede heredar directamente de una clase Contexto, pero esto liga el comportamiento a dicho Contexto, mezclando la implementación del algoritmo con la del Contexto. Además que no permite la modificación del algoritmo dinámicamente.

3. *Las estrategias eliminan las sentencias condicionales.* Ofrece una alternativa a la sentencias condicionales para seleccionar el comportamiento deseado.

4. *Una elección de implementaciones.* Las estrategias pueden proporcionan distintas implementaciones para un mismo comportamiento.

5. *Los clientes deben conocer las distintas estrategias.* El cliente debe comprender como difieren las Estrategias antes de seleccionar una adecuada. Los clientes pueden estar expuestos a cuestiones de implementación.

6. *Costes de comunicación entre Estrategia y Contexto.* La interfaz de Estrategia es compartida por todas las clases EstrategiaConcreta, ya sea el algoritmo trivial o complejo. Esto hace que sea probable que algunas estrategias no utilicen toda la información que reciben a través de dicha interfaz, por lo que el contesto crea e inicializa parámetros que nunca se usan.

7. *Mayor número de objetos.* Las estrategias aumentan el número de objetos de una aplicación.

8.3.10 Template Method

8.3.11 Visitor

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

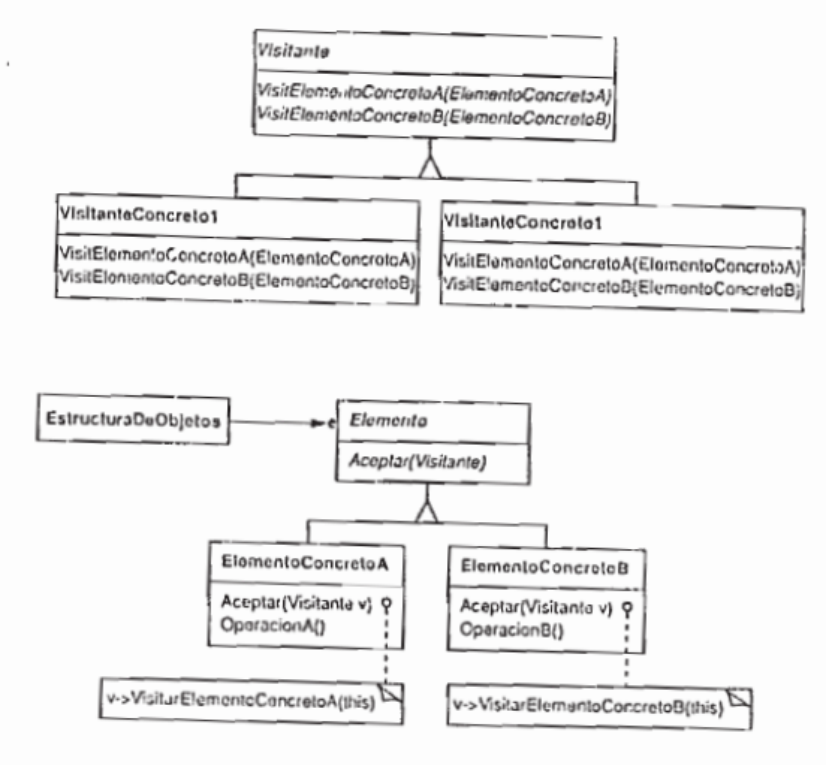
Aplicabilidad. Use el patrón cuando:

- Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre estos elementos que dependen de su clase concreta.

- Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y queremos evitar contaminar sus clases con dichas operaciones. Cuando la estructura de objetos es compartida por varias aplicaciones, el patrón permite poner operaciones sólo en aquellas aplicaciones que las necesiten.

- Las clases que definen la estructura de objetos rar vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura. Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es potencialmente costoso. Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases.

Estructura.



Participantes.

- **Visitante**
  - Declara una operación Visitar para cada clase de opreación ElementoConcreto de la estructura de objetos. El nombre y signatura de la operación identifican a la clase que envía la petición Visitar al visitante. Eso permite al visitante determinar la clase concreta de elemento que está siendo visitada. A continuación el visitante puede acceder al elemento directamente a través de su interfaz particular.
- **VisitanteConcreto**
  - Implementa cada operación declarada en Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura. VisitanteConcreto proporciona el contexto para el algoritmo y guarda su estado local. Muchas veces este estado acumula resultados durante el recorrido de la estructura.
- **Elemento**
  - Define una operación Aceptar que toma un visitante como argumento
- **ElementoConcreto**
  - Implementa una operación Aceptar que toma un visitante como argumento.
- **EstructuraDeObjetos**
  - Puede enumerar sus elementos.
  - Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.
  - Puede ser un compuesto o una colección, como una lista o conjunto.

Colaboraciones.

- Un cliente que usa el patrón Visitor debe crear un objeto VisitanteConcreto y a continuación recorrer la estructura, visitando cada objeto con el visitante.

- Cada vez que se visita a un elemento, éste llama a la operación del Visitante que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.

Consecuencias.

- *El visitante facilita añadir nuevas operaciones.* Los visitantes facilitan añadir nuevas operaciones que dependen de los componentes de objetos complejos. Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante.
- *Un visitante agrupa operaciones relacionadas y separa las que no lo están.*
- *Es dificil añadir nuevas clases de ElementoConcreto.* Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación en cada clase VisitanteConcreto. Por tanto, la cuestión fundamental a considerar a la hora de aplicar el patrón Visitor es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen a la estructura.
- *Visitar varias jerarquías de clases.* Puede visitar objetos que no tienen una clase padre en común. Se puede añadir cualquier tipo de objeto a la interfaz de un Visitante.
- *Acumular el estado.*
- *Romper encapsulación.* El patrón asume que la interfaz de ElementoConcreto es lo bastante potente como para que los visitantes hagan su trabajo. Como resultado el patrón suele obligarnos a proporcionar operaciones públicas que accedan al etado interno de un elemento, lo que puede romper su encapsulación.

8.4 Como documentar el uso de Patrones de Diseño

Pattern	**EstructuraDocumento**
based on	Composite
because	
	Cambios previstos:
	Funcionalidad:
	Restricciones de diseño:
where	Componente **is** Glifo Compuesto **is** Fila Hoja **is** Caracter Hoja **is** Rectangulo Hoja **is** Poligono operacion() **is** dibujar() operacion() **is** interseca() operacion() **is** limites() anadir() **is** insertar() eliminar() **is** borrar() obtenerHijo() **is** hijo() obtenerPadre() **is** padre() hijos **is** hijos
comments	

9 Estilos Arquitectónicos

9.1 Invocación Implícita

También conocido como *Tool Abstraction, Eventos y Publicar/Suscribir*

9.1.1 Propósito

Dos de las desventajas del **DOO** son:

1. Para que un objeto pueda invocar los servicios de otro objeto el primero debe tener una referencia a este último.
2. Respecto del invocante, los servicios ofrecidos por un objeto están fijos en tiempo de compilación.

Este estilo arquitectónico elimina ambas desventajas cambiando el conector del **DOO** (llamada a procedimiento) por el conector evento. Por lo tanto, el propósito de este estilo es permitir a los objetos invocar servicios de otros objetos sin necesidad de conocer sus identidades y permitir que, para los clientes de un objeto, las subrutinas en su interfaz no queden fijas en tiempo de compilación.

9.1.2 Aplicabilidad

Se sugiere aplicar este estilo cuando:

1. Se quiera mantener desacoplados a los componentes del sistema; en el caso extremo se espera que ningún componente sepa de la existencia de otros componentes.
2. Los componentes no requieran pasarse grandes cantidades de información entre sí.
3. No se está seguro que las interfaces de los componentes sean las que actualmente están definidas y se espere se requieran cambios en ellas.
4. Se quiera mantener muy independientes a los distintos sub-sistemas de un sistema; posiblemente los sub-sistemas implementen otros estilos.

9.1.3 Componentes

El estilo contempla dos clases de componentes: TADs y toolies. En cualquier caso los componentes tienen dos interfaces: en una anuncian eventos y en la otra exportan subrutinas que pueden ser invocadas por llamada a procedimiento. Las toolies se diferencian de los TADs en lo siguiente:

1. Los TADs se definen primero. Es decir, metodológicamente, los primeros módulos definidos por el ingeniero son TADs.
2. Los TADs son más grandes y complejos.
3. Los TADs representan las abstracciones más estables y claves del sistema.
4. Las toolies implementan requisitos secundarios y/o temporales.
5. Las toolies implementan FSM de muy pocos estados.
6. El tiempo de vida de una toolie suele ser menor al de un TAD.

Por otro lado, la infraestructura de un sistema puede incluir de forma implícita o explícita un administrador de eventos el cual puede considerarse un componente más de distinto tipo a los dos ya descriptos. Es muy probable que este componente venga dado por el entorno donde ejecutará o se desarrollará el sistema.

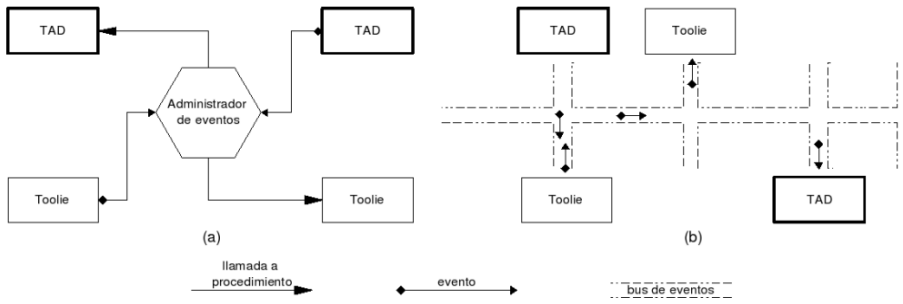
9.1.4 Conectores

1. Eventos.

Existen dos formas de comunicar los eventos entre los componentes:

1. Mediante un **bus** de eventos. Cada componente que anuncia un evento lo hace poniendo el evento en el bus. El bus transporta el evento por broadcast a todos los componentes; sólo los interesados lo toman.

2. Mediante el administrador de eventos. Este se comunica con el resto de los componentes vía llamada a procedimiento pero esto debería ser transparente para el equipo de desarrollo.



9.1.5 Patrones Estructurales

Los TADs y las toolies interactúan entre sí sólo por intermedio del administrador de eventos o poniendo un evento en el bus de eventos.

El administrador de eventos invoca procedimientos en la interfaz de los TADs o las toolies. Se asume que ninguna de las llamadas efectuadas por el administrador de eventos producirá resultados; es decir, ningún TAD o toolie debe retornar datos al administrador de eventos.

Existen dos patrones estructurales posibles dependiendo si se piensa en un administrador de eventos o en un bus de eventos, como se muestra en la Figura.

9.1.6 Modelo Computacional Subyacente

En el caso de que se piense en un administrador de eventos, el modelo computacional subyacente es el siguiente:

- Cada TAD y toolie suscribe, ante el administrador de eventos, uno o más de los procedimientos en su interfaz para uno o más eventos.
- Cuando un TAD o toolie anuncia un evento, el administrador de eventos lo recibe y lo distribuye a todos los suscriptores de ese evento.
- El administrador de eventos instrumenta la distribución de eventos haciendo llamadas a procedimiento a las subrutinas suscritas para cada evento.

El modelo computacional subyacente en el caso en que se piense en un bus de eventos es el siguiente:

- Los componentes anuncian eventos poniéndolos en el bus de eventos.
- El bus de eventos utiliza un mecanismo de *broadcast* para comunicar cada evento anunciado. Por tanto, todos los componentes conectados al bus recibirán el evento.
- Cada componente decide si está o no interesado en cada uno de los eventos que son anunciados. En caso de que lo esté, el componente decide cómo reaccionar.

Notar que de esta forma ningún componente del sistema sabe de la existencia de otros componentes ni de los servicios exportados por aquellos.

9.1.7 Invariantes Esenciales

- Ningún TAD o toolie conoce la existencia, identidad o servicios de otro TAD o toolie.
- Los componentes que anuncian eventos no saben qué otros componentes serán afectados en cada anuncio; en otras palabras ningún componente puede conocer la lista de suscriptores a los eventos que anuncia.
- Los suscriptores no pueden asumir ningún orden para la aparición de los eventos que les interesan ni que estos alguna vez aparecerán.

9.1.8 Metodología de Diseño

1. Definir los **TADs**. Se deben definir las subrutinas y los eventos (anunciados) en la interfaz de cada TAD. Cada TAD representa, implementa o es responsable del comportamiento de una de las abstracciones claves del sistema. Se los debe elegir de forma tal que cada uno implemente la funcionalidad más estable de la abstracción que representa. Dentro de esta fase podemos distinguir dos etapas:
   - Definir las subrutinas del TAD. (DOO)
   - Definir los eventos que anunciará el TAD. Esta etapa requiere planificar con mucho cuidado los eventos que anunciará cada TAD. Estos eventos tienen que estar orientados a permitir la inclusión, modificación o eliminación de ciertas funciones del TAD sin que sea necesario modificar su implementación.
2. Definir las **toolies**. Estos componentes sirven para implementar nuevos requerimientos, variantes de los servicios provistos por los TADs, implementar los requisitos secundarios del sistema o mantener relaciones entre los TADs.
3. Definir la **configuración del administrador de eventos.** Dependiendo de la especialización elegida, al configuración del administrador de eventos puede hacerse en tiempo de compilación o en tiempo de ejecución. Pero, en cualquier caso, es independiente de la implementación y definición de los TADs y las toolies. Definir la configuración del administrador de eventos significa suscribir subrutinas ante el administrador de eventos para ciertos eventos.

9.1.9 Ventajas y Desventajas

Debido al escaso acoplamiento entre los componentes del sistema tenemos que los sistemas basados en este estilo suelen presentar:

- Mayor reuso dado que los componentes no dependen del contexto porque no tienen referencias externas a otros componentes.
- Facilita la evolución del sistema. Permiten modificar, agregar y eliminar funcionalidad sin tener que modificar los componentes existentes.
- Permiten alterar fácilmente el algoritmo general de procesamiento.
- Permiten modificar las estructuras de datos claves del sistema sin afectar a los otros componentes del sistema.

Por su naturaleza, este estilo:

- Torna complicado, en algunos casos, predecir el funcionamiento del sistema.
- Suele impedir la transmisión de grandes volúmenes de datos.

9.1.10 Documentación

- **Guía de Módulos.** En la descripción de cada módulo se agrega los eventos que anuncia (dando y explicando sus parámetros, si los hubiera, la semántica de cada evento y las condiciones bajo las cuales se emite) y los eventos en que el módulo está interesado.
- **Configuración del Adm. de Eventos.** Si la suscripción a eventos es estática (ver Especializaciones) la configuración se incluye en la documentación 2MIL (ver ítem siguientes); si la suscripción es dinámica se deberá incluir un nuevo documento (generalmente en forma de tabla) donde se muestra la configuración hasta donde es posible o se explica cómo se espera que esta configuración varíe (pues en casos extremadamente dinámicos no será posible saber con antelación qué componentes se suscribirán a qué eventos).
- **Especialización Seleccionada.** Es un documento coloquial que describe todas los puntos de la especialización seleccionada.
- Extendemos **2MIL** agregando la cláusula announces donde se listan los eventos anunciados por el módulo. Si los eventos tienen parámetros estos se listan aquí dando su nombre y tipo.
- Si parte de la configuración del administrador de eventos es estática extendemos **2MIL** agregando la cláusula *callonevent* que lista los suscriptores de este módulo para ciertos eventos. Es decir, la cláusula lista pares de la forma *(eveto, subrutina)* donde evento es un evento anunciado por algún otro componente y subrutina es una de las subrutinas en el interfaz de este módulo. Más precisamente la cláusula tiene la siguiente sintaxis:

  $$\textbf{CALLONEVENT}\ evento\ p_1,\ \ldots\ , p_n\ \ \textbf{CALLS}\ subr(p_{i1},\ \ldots, p_{ik})$$

  donde $p_i\ i = 1, ..., n$ son los posibles parámetros del evento y $p_{in}\ n = 1, ..., k$ es un subconjunto de esos parámetros. En este caso los parámetros del evento se utilizan para mostrar cuáles y cómo son pasados a la subrutina invocada. Más suscripciones se pueden agregar en la misma cláusula separándolas con comas o escribiéndolas en líneas separadas.

  Puede ocurrir que el evento no tenga parámetros y la subrutina suscrita sí los tenga. En estos casos se deberán indicar las constantes que se le pasarán a la subrutina.

9.1.11 Especializaciones comunes

Las especializaciones de este estilo surgen al combinar diferentes alternativas en cuatro grandes áreas. Las alternativas más comunes o más recomendadas aparecen en negrita.

- **Definición de eventos.** Refiere al vocabulario de eventos que manejará el sistema. Las alternativas son:

  1. Vocabulario fijo de eventos. No es posible definir nuevos eventos. El sistema provee un cierto conjunto de eventos y solo se pueden usar esos.
  2. **Declaración estática de eventos.** El programador puede declarar nuevos eventos pero debe hacerlo en tiempo de compilación, es decir, no es posible añadir eventos en tiempo de ejecución.
  3. No hay declaración de eventos. Los eventos no se declaran y por lo tanto se pueden agregar eventos en cualquier momento. Si bien esta alternativa es la más flexible se considera que puede generar sistemas poco predecibles e indisciplinados.
  4. Declaración centralizada de eventos. Los eventos se declaran y se declaran todos en el mismo módulo del sistema.
  5. Declaración distribuida de eventos. Los eventos se declarar pero cada evento se lo puede declarar en un módulo diferente al resto.

- **Estructura de los eventos.** Refiere a la posibilidad de que los eventos tengan parámetros o no. Las alternativas son:

1. Nombres simples. Los eventos no pueden tener parámetros.
2. Lista de parámetros fija. Todos los eventos tienen la misma cantidad y tipo de parámetros.
3. **Parámetros por tipo de evento.** Cada evento (no cada anuncio de un evento) puede tener su propia lista de parámetros. En este sentido podría pensarse que cada nombre de evento define un tipo de evento (poblado por todos los anuncios de ese evento) y entonces decimos que cada tipo de evento tiene su propia lista de parámetros.
4. Lista de parámetros dependiente del anuncio. Cada vez que se anuncia un evento de cierto tipo la lista de parámetros puede diferir de los otros anuncios. Nuevamente esto es más flexible pero también más propenso a errores, complicaciones, etc.

- **Asociación de eventos.** Refiere al momento en el cual se establecen las suscripciones y la forma de pasar los parámetros de los eventos a los suscriptores. Las alternativas son:

  1. **Asociación estática.** Los subscriptores se establecen en tiempo de compilación. Es decir el programador establece la configuración del sistema en el momento de la compilación.
  2. Asociación dinámica. Los suscriptores se establecen en tiempo de ejecución lo que da nuevamente mayor flexibilidad pero también crea sistemas menos predecibles. Un suscriptor puede desuscribirse o suscribirse a uno o varios eventos a medida que el sistema ejecuta.

  También es importante determinar cómo se pasarán los parámetros de los eventos (si los tuvieran) a los procedimientos suscritos. Las alternativas son:

  1. Pasaje total de parámetros. La cantidad y tipo de los parámetros del evento deben coincidir con los de cada uno de sus suscriptores. En este caso cada parámetro real del evento se pasa al correspondiente parámetro formal del procedimiento suscrito.
  2. **Pasaje selectivo de parámetros.** El programador o el administrador (según haya asociación estática o dinámica) pueden definir cuáles de los parámetros del evento desean recibir y a cuáles de los parámetros formales del suscriptor corresponde cada uno.
  3. Pasaje según el resultado de expresiones. En esta alternativa cuáles parámetros del evento se pasarán y a qué parámetros formales corresponden se decide en tiempo de ejecución y en función del resultado de expresiones.

- **Política de entrega.** Refiere a la forma en que el administrador de eventos invoca a los suscriptores ante la aparición de un evento. En la mayoría de los sistemas basados en II los eventos son anunciados a todos sus suscriptores (lo que implica que se ejecutan las subrutinas asociadas). Las alternativas que presentamos son:

  1. **Entrega completa.** Un evento es anunciado a todos sus suscriptores.
  2. Entrega simple. Cada evento es manejado por un único suscriptor (si es que hay uno). Es útil, por ejemplo, cuando hay varios suscriptores pero semánticamente solo uno debe ser invocado (Por ej. cuando se anuncia un evento como "tomar bulto", al cual pueden estar suscritos varios brazos de un robot, sólo el primero que esté libre debe reaccionar).
  3. Selección basada en parámetros. Esta alternativa utiliza los parámetros del evento anunciado para decidir cuáles de todos sus suscriptores deben ser invocados. En consecuencia dos anuncios del mismo evento (aunque con parámetros diferentes) pueden implicar invocaciones distintas.
  4. Política de entrega por evento. Se asocia una política de entrega con cada evento. En el momento en que el evento aparece, el sistema determina, en base a la política definida para ese evento, el efecto que tendrá el evento: anunciarlo, no anunciarlo, disparar otros eventos, etc. Esta alternativa provee muchas de las ventajas de los sistemas dinámicos sin ser tan complejo.

9.1.12 Deformaciones Comunes

Obviamente la deformación más común del estilo es combinarlo con llamada a procedimiento explícita. En muchos casos esto es muy conveniente sobre todo para comunicar grandes cantidades de datos. Debería restringirse a los componentes cuyas interfaces se consideran prácticamente inamovibles, los que por lo general son los TAD.

9.2 Tubos y Filtros

9.2.1 Propósito

Este estilo arquitectónico provee la estructura y los mecanismos para los sistemas que deben procesar flujos de datos. Cada etapa del procesamiento es encapsulada en un filtro. Los datos se transmiten a través de tubos entre filtros adyacentes. Se pueden obtener familias de sistemas relacionados recombinando, eliminando y agregando filtros.

9.2.2 Aplicabilidad

Se sugiere aplicar este estilo en los siguiente casos:

1. Procesamiento de señales.
2. Procesamiento de imágenes o sonido.
3. Compiladores.
4. Procesamiento de cadenas.
5. Sistemas con poca o nula interacción con el usuario cuyo flujo de datos se entienda o perciba como continuo (es decir en forma de stream).
6. También se menciona la posibilidad de aplicarlo para el cálculo de la Transformada de Fourier Discreta (Rápida), algoritmos de búsqueda en paralelo y modelos de simulación científica.

9.2.3 Componentes

Los componentes de este estilo se denominan **filtros**. Los filtros son las unidades de procesamiento del sistema. Cada filtro enriquece, refina o transforma sus datos de entrada produciendo un flujo de salida. La interfaz de cada filtro consta de un conjunto de puertos de entrada y un conjunto de puertos de salida; ambos son conjuntos no vacíos y disjuntos. Los datos de entrada arriban al filtros en sus puertos de entrada en tanto que este pone a disposición del entorno el resultado de su procesamiento en los puertos de salida.

9.2.4 Conectores

El único mecanismo de interacción disponible para los filtros se denomina **tubo**. Un tubo puede conectar dos filtros o un filtro con su entorno (entrada o salida estándar, por ejemplo). Un tubo tiene dos extremos. Cuando un extremo de un tubo se conecta a filtro se lo conecta a uno de sus puertos. La única función del tubo es llevar los datos que entran por uno de sus extremos al otro extremo sin modificarlo de ninguna manera (al menos de ninguna manera perceptible).

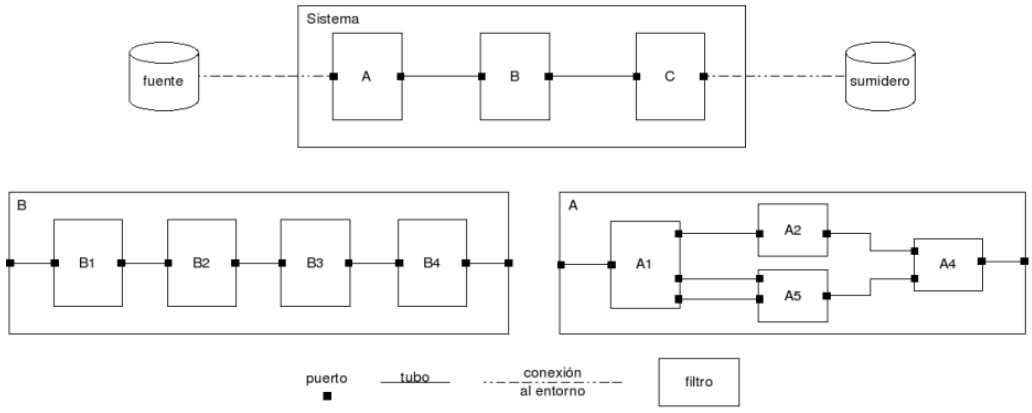
9.2.5 Patrones Estructurales

Un sistema de tubos y filtros forma un grafo dirigido acíclico desde la **fuente** de datos hacia su **sumidero**. La fuente (entrada estándar, por ejemplo) es uno o varios componentes externos que proveen los datos de entrada al sistema; el sumidero (salida estándar, por ejemplo) son uno o varios componentes externos que reciben los datos emitidos por el sistema. Los datos fluyen desde la fuente, pasando por el sistema de tubos y filtros, hacia el sumidero. Siempre supondremos que el flujo de información va de izquierda a derecha.

9.2.6 Modelo Computacional Subyacente

Cada filtro computa independientemente de los demás; puede hacerlo en cuanto tiene datos disponibles en alguno de sus puertos de entrada. Lo más usual es que cada filtro revise periódicamente sus puertos de entrada en busca de datos y si los hay los consuma y procese, produciendo más tarde un flujo de salida sobre uno o más de sus puertos de salida; este tipo de filtros se llaman **activos**. Otra alternativa es que los tubos utilicen la interfaz de los filtros para comunicar o tomar los datos; en este caso los filtros son **pasivos**.

Una vez que un tubo detecta la presencia de datos en su extremo izquierdo los traslada a su extremo derecho. Cuando el extremo izquierdo está conectado a un filtro, el tubo detecta la presencia de datos cuando aquel los pone en el puerto de salida correspondiente.



9.2.7 Invariantes Esenciales

Los invariantes del estilo son los siguientes:

- Ningún filtro conoce la identidad ni la existencia de otros filtros en el sistema.
- Ningún filtro depende o conoce el estado de los otros filtros del sistema.
- La corrección de la salida del sistema no depende del orden en que cada filtro realice sus cálculos.
- Los tubos no realizan ningún tipo de modificación sobre los datos que transmiten

9.2.8 Metodología de Diseño

La metodología de diseño más común a la hora de aplicar este estilo es la descripta en los siguientes pasos.

1. *Dividir la tarea del sistema en una secuencia de etapas de procesamiento.* Cada etapa debe depender únicamente de la entrada que recibe. Apuntar a obtener una división jerárquica (composicional).
2. *Definir el formato de datos a transmitir por cada puerto/tubo.* Definir un formato uniforme para todo el sistema implica tener mayor flexibilidad porque hace que la recombinación de filtros sea más simple. Sin embargo, puede presentar problemas de desempeño y puede llevar a una organización más compleja o caótica del sistema.
3. *Decidir cómo implementar las conexiones de los tubos a los filtros.* Usualmente se utiliza algún mecanismo por el cual los filtros producen y consumen cantidades diferentes de datos de los tubos. A su vez cada tubo provee alguna forma de almacenamiento temporario (buffering) y sincronización entre productor y consumidor. También hay que determinar el comportamiento de bloqueo cuando el tubo está lleno y se intenta escribir y cuando está vacío y se intenta leer.
4. *Diseñar e implementar los filtros.* Se consideran las etapas de procesamiento del paso 1. Para cada filtro se listan sus puertos de entrada y salida (y el tipo de cada uno si lo hubiera) y se da una especificación funcional de la máquina de estados que implementará.
5. *Diseñar el manejo de errores.* Dado que un sistema de TyF no comparte un estado global el manejo de errores es muy difícil y muchas veces se lo niega. Al menos debe existir una forma de detección de errores.

9.2.9 Ventajas y Desventajas

Dentro de la ventajas de utilizar este estilo tenemos las siguientes:

- Alienta y habilita a pensar el sistema como la **composición funcional** de filtros. En efecto, un sistema de tubos y filtros es en sí mismo un filtro.
- **Reutilización.** Dado que los filtros no dependen de otros filtros, cada uno de ellos puede ser reemplazado por uno o más filtros, puede ser eliminado, otros filtros pueden ser agregados, etc.

  Por otro lado, un filtro desarrollado para un sistema podrá ser utilizado en cualquier otro sistema donde se transmitan los mismos tipos de datos.

  Esta propiedad se obtiene a partir de la gran independencia con respecto al contexto de cada filtro.
- **Bajo acoplamiento.** Precisamente la escasa dependencia del contexto es sinónimo de bajo acoplamiento. Por lo tanto, los filtros verifican uno de los principios básicos del diseño: componentes altamente cohesivos y escasamente acoplados.
- **Concurrencia.** El estilo se presta naturalmente para implementaciones concurrentes. En el caso más extremo cada filtro podría correr en un procesador dedicado únicamente para él. Sin embargo, aquellos filtros que solo puedan emitir una vez recibida toda la entrada (por ejemplo un filtro de ordenación), retrasarán el funcionamiento del sistema en general; es decir se convierten en cuellos de botella. Al mismo tiempo, la descripción arquitectónica del sistema permite detectar tempranamente estos cuellos de botella y tomar medidas paleativas.
- El **bajo acoplamiento** permite asegurar la calidad de cada filtro con gran independencia de los demás.
- El **bajo acoplamiento** habilita la posibilidad de desarrollar independientemente cada uno de los filtros.
- El estilo **facilita el análisis del rendimiento de procesamiento** (throughput). En el primer caso se debe a que es relativamente simple conocer y predecir la velocidad de procesamiento de cada filtro por lo que es posible predecir y estimar la velocidad de procesamiento del sistema completo (usando composición).

Entre las desventajas tenemos:

- Organización **batch**. Dado que el problema se descompone en una secuencia de pasos, se alienta una filosofía batch. Los sistemas organizados según esta filosofía tienden a presentar graves problemas cuando se requiere interacción con el usuario.
- Debe forzarse un denominador común entre los datos transmitidos. Esto suele implicar la necesidad de que varios filtros analicen la entrada para detectar los elementos con los cuales necesitan trabajar y volver a generar una secuencia del tipo a transmitir antes de emitir la salida. Todo esto redunda en una pérdida de desempeño.
- Si un filtro no puede producir salida hasta no haber recibido toda la entrada, el filtro requerirá un buffer de tamaño arbitrario. El sistema podría entrar deadlock si los buffers tienen tamaño fijo.
- Pérdida de desempeño al tener que copiar varias veces el mismo dato. Un filtro que, por ejemplo, elimina ciertos caracteres debe copiar del puerto de entrada a su espacio de memoria la cadena completa y luego copiar la cadena filtrada al al puerto de salida; mientras que en otro estilo hubiera sido posible inspeccionar la cadena y borrar los elementos no deseados sin tener que realizar ninguna copia.

9.2.10 Documentación

La documentación específica para diseños basados en TyF es la siguiente:

- **Diagrama canónico o configuración.** El diagrama canónico de un diseño basado en TF es la configuración del sistema, es decir se listan las instancias de los filtros (puede haber más de una instancia del mismo filtro) que formarán el sistema y se consignan todas las conexiones.

  Usualmente esto puede hacerse con un grafo. Siempre se asume que el flujo de datos va de izquierda a derecha y, por lo tanto, los puertos de entrada de un filtro son los del lado izquierdo y los de salida los del lado derecho.

  La representación gráfica también puede hacerse textualmente.

  Tener en cuenta las siguientes notas:

  - Los filtros deben estar nombrados (el nombre puede coincidir con su "tipo" o no).
  - Los filtros se dibujan todos con el mismo símbolo.
  - Los puertos se dibujan todos con el mismo símbolo.
  - Los tubos se dibujan con una línea continua (no hace falta indicar una flecha dada la suposición anterior).
  - Las conexiones con la fuente y el sumidero se grafican con una línea de puntos (no hace falta indicar una flecha dada la suposición anterior).
  - Si se usan sintonizadores se grafican con una flecha que apunta al lado superior (o inferior si no hay suficiente espacio) del filtro.
  - El "tipo"de cada filtro se escribe dentro de la caja del filtro correspondiente.
  - En lo posible se nombran los puertos. Si el dibujo se torna muy complicado se pueden numerar y referenciar en otra parte.

- **Especialización/Deformación seleccionada.** Documento que describe las cuestiones de diseño que quedan abiertas en la descripción del estilo. Por ejemplo: uso de sintonizadores, comportamiento de bloqueo en tubos, implementación como procesos o hilos de ejecución, manejo de errores, etc.

- Se agregan las cláusulas **INPORTS** y **OUTPORTS**:

  INPORTS *nombre* : [*tipo*]
  OUTPORTS *nombre* : [*tipo*]

  Si se usan sintonizadores se incluye la cláusula exports en la cual cada sintonizador es un procedimiento.

- Por lo general, los tubos se documentan como sigue.

Generic Module	**Tubo(X)**
exportsproc	read():X write(**i** X)
comments	X puede reemplazarse por *array(X)* si se quiere una lectura/escritura de mayor longitud.

  El parámetro **X** sirve también para el caso en que se usen tubos tipados.

- **Guía de Módulos.** Función de cada filtro en términos de sus puertos de entrada y salida. Si se usó composición para dar una estructura jerárquica de filtros, la estructura de la guía debe reflejarlo.

  Documentar los módulos que corresponden a los tubos.

- **Estructura de Módulos.** Tener en cuenta el uso de composición de filtros.

9.2.11 Especializaciones Comunes

Las dos especializaciones más comunes son las siguientes. Ambas pueden combinarse.

- **Pipelines.** En esta especialización todos los filtros tienen un único puerto de entrada y un único puerto de salida. Para muchos casos esto suele ser suficiente.

- **Puertos y tubos tipados.** Cada puerto, y en consecuencia el tubo que a este se conecta, transmite un tipo de dato (posiblemente) diferente al de otros puertos, incluso del mismo filtro. Entonces se dice que el puerto tiene un tipo o es de cierto tipo; lo mismo ocurre con los tubos. De esta forma, si un puerto tiene tipo T solo podrá conectarse a un tubo del mismo tipo. El tubo tiene el mismo tipo en sus dos extremos (pues de otra forma implicaría que el tubo lleva a cabo una conversión de tipo violando uno de los invariantes esenciales del estilo).

  Si bien esta especialización puede dar lugar a menor flexibilidad (ya que ahora los filtros dependen más del contexto que antes) permite mejorar la eficiencia en la transmisión de información pues no requiere que los filtros hagan una y otra vez conversiones de datos.

  Además, reduce la posibilidad de errores de programación al habilitar un uso más extendido de tipos.

  Finalmente, en el mismo sentido, permite transmitir tipos complejos, que pueden implementarse como TADs u objetos lo que agrega aun más orden al aplicar el principio de ocultación de información.

9.2.12 Deformaciones Comunes

Algunas de las deformaciones intentan reducir las desventajas del estilo; otras agregan más posibilidades pero ponen en riesgo la corrección del sistema.

- **Repositorio común para los datos.** Una de las desventajas del estilo es la pérdida de rendimiento debido a que es necesario copiar (en general muchas veces) un mismo dato. Por este motivo, en los casos en que esa pérdida sea permitida, se sugiere utilizar un repositorio común de datos.

  En este caso todos los filtros obtienen sus datos de un repositorio de datos y guardan allí el resultado de su procesamiento. El procesamiento puede llevarse a cabo en el repositorio mismo; esto violaría el principio de ocultación de información. Si se usa un repositorio común deberá existir un componente o mecanismo que permita coordinar el acceso a los datos comunes. Aun así ningún filtro debe conocer la identidad de los otros.

  El repositorio no necesariamente debe reemplazar a todos los tubos, ambos mecanismos de comunicación pueden utilizarse según convenga.

- **Sintonizadores.** Otra de las desventajas del estilo es la imposibilidad de interactuar con el usuario. Para mejorar este aspecto se puede incluir otra interfaz en los filtros que sea necesario. Esta interfaz está compuesta por uno o más **sintonizadores**. Un **sintonizador** es un procedimiento que permite alterar o sintonizar el comportamiento del filtro. Por ejemplo, un filtro que renderice imágenes podrá tener un sintonizador para ajustar con precisión el algoritmo. Los sintonizadores solo pueden ser invocados por componentes que no sean filtros del sistema; generalmente por un componente que implementa una GUI.

- **Ciclos.** El patrón estructural básico impide la existencia de ciclos en la red de tubos y filtros. Una posibilidad para obtener sistemas más generales es eliminar esa restricción y en consecuencia permitir la existencia de ciclos. En este caso el diseño exige un fundamente muy sólido para explicar y entender el cálculo completo; un análisis teórico riguroso y el uso de especificaciones formales son apropiados para demostrar que el sistema termina y que produce el resultado deseado.

9.3 Sistemas Estratificados

También conocido como *Máquinas abstractas jerárquicas.*

9.3.1 Propósito

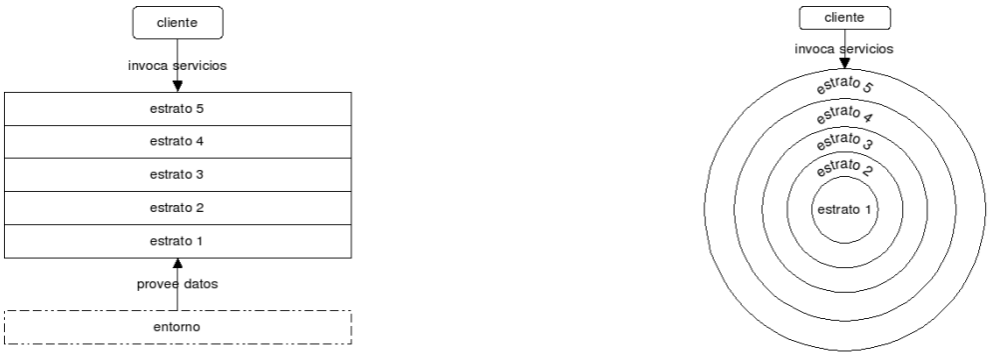
Este estilo arquitectónico es muy útil para estructurar aplicaciones que pueden ser descompuestas en grupos de sub-tareas cada una de las cuales está a un nivel de abstracción particular. Los grupos están ordenados jerárquicamente según su nivel de abstracción.

9.3.2 Aplicabilidad

- Máquinas Virtuales (Java Virtual Machine, por ejemplo)
- APIs.
- Sistemas operativos.
- Protocolos de red (por ejemplo el modelo OSI y en particular la forma en que se implementan los protocolos TCP/IP).
- Ciertas lógicas de negocio o partes de ellas (por ejemplo, el caso de los medios de pago del software para controlar la estación de peaje visto anteriormente).

9.3.3 Componentes

Los componentes de este estilo se denominan **estratos**. Un estrato ofrece un conjunto de subrutinas en su interfaz y realiza llamadas a la interfaz de otros estratos. Cada estrato debe proveer una funcionalidad más abstracta o con una semántica más rica o más orientada al negocio del sistema, que la provista por aquellos estratos a los cuales invoca. Un estrato puede estar estructurado en módulos que en conjunto proveen los servicios del estrato. Estos módulos cooperan entre sí para proveer la funcionalidad del estrato.



Los estratos pueden estar estructurados en módulos.

9.3.4 Conectores

El conector fundamental es llamada a **procedimiento**. Un estrato invoca los servicios de otro estrato por medio de llamada a procedimiento. Los módulos que componen un estrato pueden conectarse también por llamada a procedimiento.

Es común también que se usen eventos para comunicar información desde los estratos inferiores hacia los superiores.

9.3.5 Patrones Estructurales

Como ya se mencionó cada estrato está a un nivel de abstracción particular. La restricción estructural fundamental en los sistemas estratificados es que si un estrato provee servicios en el nivel de abstracción k solo podrá invocar servicios del estrato que corresponde al nivel de abstracción $k-1$.

En cualquier caso, ya sea que los estratos estén estructurados por más de un módulo o no, el estrato que provee la funcionalidad más abstracta es el superior o externo y el que provee la funcionalidad menos abstracta es el inferior o interior.

9.3.6 Modelo Computacional Subyacente

Existen varias posibilidades en el comportamiento dinámico del sistema; las dos más comunes se describen a continuación.

- **Desde el Cliente.** Un cliente requiere un servicio del estrato superior, el cual al no poder resolverlo con sus propios recursos invoca uno o más servicios del estrato inmediato inferior. Esta cadena de invocaciones continúa en cascada posiblemente hasta el estrato inferior.
- **Desde el entorno.** Puede ocurrir que el estrato inferior detecte una alteración en el entorno (o este le comunica tal alteración) que debe ser transmitida al estrado inmediato superior. Como ningún estrato puede invocar servicios de los estratos superiores, entonces es necesario incluir algún mecanismo de comunicación indirecta o implícita como *callbacks* o eventos. Esta cadena de invocaciones puede ascender hasta el estrato superior.

  En el caso en que el mecanismo indirecto utilizado sea eventos, los suscriptores a un evento anunciado por el estrato k debe ser únicamente subrutinas del estrato $k+1$. Estas subrutinas u otras del estrato $k+1$ pueden invocar servicios provistos por k para obtener más información.

9.3.7 Invariantes Esenciales

- El estrato k sólo puede invocar servicios del estrato $k-1$.
- El estrato k no sabe de la existencia del estrato $k+1$.
- Todos los componentes que forman un estrato están al mismo nivel de abstracción entre sí.

9.3.8 Metodología de Diseño

1. **Definir el criterio de abstracción para agrupar funciones en estratos.** Usualmente es la distancia conceptual desde la plataforma; también puede ser el grado de complejidad conceptual.
2. **Determinar el número de niveles de abstracción** siguiendo el criterio establecido en el paso 1. Normalmente cada nivel de abstracción corresponde a un estrato pero hay caso en los cuales no es trivial decidir si dos niveles de abstracción deben unirse en un único estrato.
3. Nombrar los estratos y determinar la funcionalidad de cada uno.
4. **Especificar los servicios de cada estrato** (no necesariamente la interfaz). Ningún módulo o componente puede abarcar más de un estrato. Los valores de retorno y errores de las subrutinas ofrecidas por el estrato k deberían estar construidos en base a elementos del lenguaje de programación, tipos definidos en k o definidos en un módulo compartido entre todos los estratos.

   Suele ser mejor ofrecer más servicios en los estratos superiores y menos en los inferiores pues de esta forma los desarrolladores no tienen que conocer una gran cantidad de servicios de bajo nivel que no son muy diferentes entre sí.
5. **Refinar la estratificación.** Iterar sobre los pasos 1-4. No siempre es posible pensar o idear el criterio de abstracción sin imaginar los posibles estratos y sus servicios. No es bueno definir componentes y después pretender imponer una relación de abstracción entre ellos. Por tal motivo se sugiere iterar reiteradas veces entre los pasos 1 a 4 hasta obtener una estratificación que represente naturalmente (y no forzadamente) una relación de abstracción.
6. **Especificar la interfaz de cada estrato.** Hay tres estrategias.
   (a) **Caja Negra.** Significa que el estrato k ve al estrato $k-1$ como una caja negra, es decir, k no reconoce la existencia de módulos en $k-1$. Esto se puede lograr aplicando el patrón de diseño **Facade**.
   (b) **Caja Gris.** Significa que k sabe de la existencia de los módulos que componen a $k-1$.
   (c) **Caja Blanca.** $k-1$ no fue diseñado siguiendo el Principio de Ocultación de la Información.

7. **Estructurar cada estrato.** Históricamente se ponía énfasis en definir la estructura de estratos pero cada uno de ellos se diseñaba monolíticamente. Cuando un estrato es complejo se lo debe diseñar cuidadosamente. En general es posible aplicar la **metodología de Parnas** y patrones de diseño como **Bridge** o **Strategy**.

8. **Especificar la comunicación entre estratos adyacentes.** Usualmente todos los datos que necesita el estrato $k-1$ para cumplimentar un servicio solicitado por el estrato k se pasan como parámetro de la llamada (**push**). Otra alternativa es que k busque los datos en algún lugar específico (**pull**); sin embargo esta esquema introduce acoplamiento entre un estrato y su superior. Si se quiere mantener el esquema pull se pueden usar callbacks.

9. **Desacoplar los estratos adyacentes.** El acoplamiento producto de verificar el invariante esencial del estilo es razonable y no es necesario esforzarse para eliminarlo. Sí es muy importante eliminar el acoplamiento en el sentido contrario. Este acoplamiento puede aparecer debido a un mecanismo de comunicación tipo **pull** o por alguna estrategia para el **manejo de errores**. Entonces, para tener comunicación hacia arriba sin acoplamiento, se pueden usar **callbacks**, eventos o el patrón de diseño **Command**.

10. **Diseñar una estrategia para el manejo de errores.** Suele ser una tarea costosa tanto desde el punto de vista del desempeño como de la programación. La regla básica es manejar los errores al menor nivel posible. Se debe intentar que los errores sean significativos para los estratos superiores. Esto implica que la semántica del error debe expresarse en el nivel de abstracción de ese estrato. Como mínimo hay que tratar de condensar errores similares en un mismo error y comunicar ese.

9.3.9 Ventajas y Desventajas

Entre las ventajas tenemos las siguientes.

- Reuso de los estratos.
- Permite la estandarización.
- Las dependencias quedan confinadas (entonces tenemos portabilidad y testeabilidad).
- Intercambiabilidad.
- Subconjuntos útiles.

En tanto que algunas desventajas son las siguientes

- Cascadas de cambios de comportamiento.
- Menor eficiencia que un sistema monolítico que permita que los estratos superiores utilicen los servicios de todos los estratos inferiores.
- Procesamiento innecesario. Ocurre cuando los estratos inferiores realizan más tareas de las solicitadas o se duplican las tareas (por ejemplo
- Dificultad en establecer la granularidad correcta de los estratos.

9.3.10 Documentación

La documentación específica para SE es la siguiente.

- **Diagrama canónico.** El diagrama canónico muestra los estratos ordenados en niveles de abstracción y nombra cada uno de ellos.
- Los estratos que no tienen un diseño monolítico pueden documentarse como **módulos lógicos** que exportan una interfaz restringida respecto de las interfaces de los módulos físicos que los componen.
- Usualmente la estructura de módulos tiene un primer nivel constituido por cada uno de los estratos. Los niveles siguientes son descomposiciones más finas de cada estrato o sus componentes.

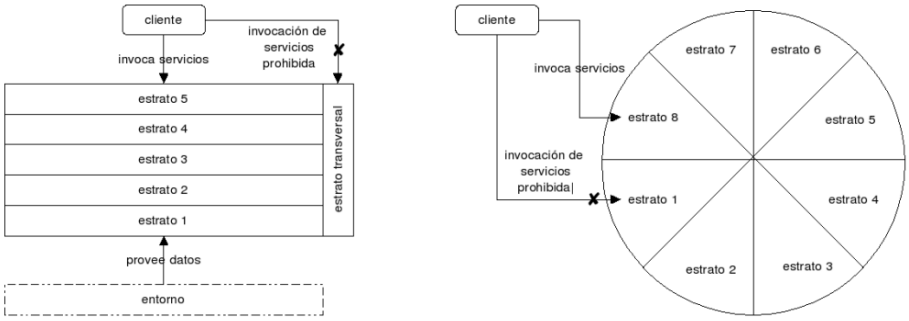
9.3.11 Especializaciones Comunes

No se han documentado.

9.3.12 Deformaciones Comunes

De una u otra forma se violan los invariantes del estilo:

- Estratos superiores pueden acceder a todos los estratos inferiores.
- Un estrato conoce a sus estratos adyacentes.
- Existen uno o más componentes compartidos por todos los estratos.



9.4 Control de Procesos

9.4.1 Propósito

Brinda la posibilidad de estructurar sistemas de control. El propósito de un sistema de control es mantener ciertas propiedades de la salida del proceso cerca de valores de referencia. El sistema de control mide esas propiedades y modifica el comportamiento de la parte del proceso encargada de producir la salida.

9.4.2 Aplicabilidad

El estilo puede aplicarse exitosamente al menos en las siguientes situaciones:

- Innumerables sistemas industriales tales como: termostatos, velocidad crucero, frenos ABS, piloto automático, celdas de producción automatizadas, etc.
- Robots móviles
- Sistemas de vigilancia automatizada
- Sistemas de detección de fallas
- Sistemas de detección de intrusos (IDS)

9.4.3 Componentes

Cualquier sistema diseñado dentro de este estilo se compone de tres componentes. Todos pueden ser diseñados como módulos abstractos o **TADs**. Desde el punto de vista del diseño no presentan características singulares sino que se distinguen por la forma en que interactúan entre sí:

- **Control.** Es el componente que implementa el algoritmo de control propiamente dicho; también presenta interfaces para activar/desactivar todo el sistema de control y establecer los rangos de funcionamiento o set points. Recibe datos proveniente de los **Sensores** y actúa sobre el **Proceso** intentando que este modifique su comportamiento de forma tal que los parámetros sensados por los sensores se mantengan dentro de los rangos de funcionamiento.

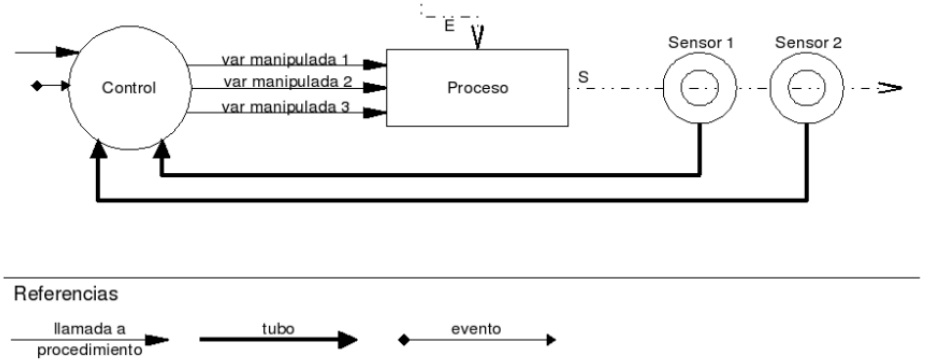
- **Proceso.** Oculta los dispositivos que producen la salida cuyos parámetros deben ser medidos y controlados. Ofrece una interfaz al **Control** de manera que pueda modificar el proceso cuando resulte conveniente. La interfaz debe ser lo suficientemente amplia como para que sea posible modificar cualquiera de las variables del proceso o variables manipuladas de forma independiente.
- **Sensores.** Miden los parámetros que deben ser controlados (se los llama variables controladas o variables medidas) y comunican esos valores al Control. Ocultan los dispositivos específicos que se utilicen para las mediciones.

9.4.4 Conectores

- **Evento.** Se utilizarán para activar/desactivar el funcionamiento del sistema de control.
- **Llamada a Procedimiento.** Se utilizarán para establecer los rangos de funcionamiento y para modificar/consultar el **Proceso**.
- **Tubo.** Se utilizarán para comunicar los datos entre los **Sensores** y el **Control**.

9.4.5 Patrones Estructurales

Existe un único patrón estructural que se muestra en la Figura Las "cajas" del diagrama representan los componentes de software y no los componentes físicos. Las líneas de puntos que entran y salen del Proceso representan la entrada y salida del proceso físico; se las incluye para mostrar que el proceso se ve influenciado por la entrada y que los sensores miden la salida producida por proceso. El componente **Proceso** no produce salida ni recibe entrada.



9.4.6 Modelo Computacional Subyacente

Desde el entorno se fijan los rangos de funcionamiento y se activa/desactiva el sistema de control. Los **Sensores** sensan las variables controladas y comunican esos valores de forma continua al **Control**. Este último analiza dichos valores, los compara con los rangos de funcionamiento y, según el algoritmo de control implementado, eventualmente invoca los servicios del Proceso para modificar su comportamiento. El **Proceso** actúa sobre uno o varios componentes externos (usualmente dispositivos físicos) según lo indica el **Control**.

9.4.7 Invariantes Esenciales

Los invariantes se derivan de los conectores utilizados y de las interacciones ya explicadas. Resumiendo:

- El **Control** es el único componente que implementa el algoritmo de control, recibe los rangos de funcionamiento y recibe los eventos para activar/desactivar el sistema.
- El **Control** desconoce desde donde provienen los valores de las variables controladas.
- El **Proceso** es el único componente que se comunica con el exterior para modificar el comportamiento del proceso que está bajo control.
- Los **Sensores** desconocen el componente que recibe los datos por ellos sensados.

9.4.8 Metodología de Diseño

La metodología de diseño al aplicar este estilo se basa en la forma en que se resuelven los problemas de control.

1. Seleccionar el principio de control. Este concepto tiene dos significados: (a) indicar en alguna forma el algoritmo de control, o (b) indicar si el control se ejercerá en ciclo abierto (**open-loop**) o ciclo cerrado (**closed-loop**) y en este último caso si será de retroalimentación (**feedback**) o de alimentación hacia adelante (**feedforward**).

2. **Seleccionar las variables del proceso** (o variables manipuladas). Estas son las variables que permiten el Control modificar el comportamiento del Proceso. Si no se sabe de qué forma se controlará al proceso no se puede comenzar a definir las fronteras de cada componente.

3. **Seleccionar las variables controladas** (o variables medidas). Estas son las variables que miden los Sensores. De igual forma que con las variables manipuladas, si no se conoce qué propiedades se deben medir y mantener dentro de los rangos de funcionamiento tampoco se podrá diseñar el sistema.

4. **Crear los sub-sistemas.** En este paso se definen las interfaces de los tres componentes básicos del diseño. Si la cantidad de variables manipuladas, o la cantidad de variables medidas son muy grandes o el algoritmo de control es muy complejo o existen varias formas de iniciar/detener el sistema de control o de fijar los rangos de funcionamiento, entonces los sub-sistemas deben dividirse en módulos más simples y manejables.

   En general un gran número de variables (medidas o manipuladas) viene asociado a un número importante de dispositivos físicos diferentes (en el proceso o en los sensores o en ambos) o muy complejos. Por lo tanto, las reglas básicas del diseño indican dividir el **Proceso** o los **Sensores** en tantos módulos como dispositivos físicos existan.

   En el mismo sentido un algoritmo de control complejo puede ser dividido en partes o los datos que utiliza pueden encapsulares en objetos o **TADs**. Igualmente, si hay varias formas de activar/desactivar el sistema de control o de fijar sus rangos de funcionamiento, en general se debe a que hay múltiples interfaces con componentes externos y por lo tanto, el módulo de control puede dividirse en: **activación/desactivación** (e incluso en varios submódulos si hay varias formas), **configuración de set points** (e incluso en varios sub-módulos si hay varias alternativas para configurarlos) y en el **algoritmo de control**. También cabe considerar la interfaz con los sensores como un sub-módulo en sí mismo.

   Posiblemente la división de estos sub-sistemas en módulos y sub-módulos pueda considerarse como parte del diseño (detallado) más que como parte de la arquitectura.

9.4.9 Ventajas y Desventajas

Este estilo clarifica el diseño de un problema que es apropiado para el estilo en varios aspectos:

- La separación entre control y proceso hace que el modelo de control sea explícito y por lo tanto más simple de verificar; de la misma forma hace que aparezca la pregunta sobre la autoridad de control.

- La existencia explícita del componente que encapsula el algoritmo de control establece la decisión de diseño sobre la clase de control que se impondrá.

- Al establecer relaciones especiales entre componentes, el estilo de CP discrimina entre diferentes tipos de entradas y hace que el ciclo de control sea más obvio.

- Permite la incorporación sencilla de más o diferentes sensores, mejoras en el algoritmo de control, cambios en los diferentes dispositivos de hardware, etc.

9.4.10 Documentación

- **Diagrama canónico.** Se documenta gráficamente como se muestra en la Figura.
- **Soporte al diagrama canónico.** Debe escribirse un documento que dé el soporte adecuado al diagrama canónico, es decir debe describir el tipo de ciclo de control diseñado (de retroalimentación o **feedback**, o de alimentación hacia adelante o **feedforward**), las variables controladas y las variables del proceso.

9.4.11 Especializaciones Comunes

No se conocen.

9.4.12 Deformaciones Comunes

No se conocen.

9.5 Blackboard Systems

9.5.1 Propósito

Organizar datos y subsistemas especializados que unen su conocimiento para calcular una solución parcial o aproximada de un problema para el cual no se conoce una solución algorítmica.

9.5.2 Aplicabilidad

El estilo BS se debe aplicar cada vez que se deba resolver un problema para el cual no se conoce una solución algorítmica (o si se conoce una, es computacionalmente muy costosa) o aquellos problemas para los cuales una solución parcial o aproximada es útil. Un ejemplo típico del primer caso es el problema del viajante, en tanto que un ejemplo típico del segundo caso es el reconocimiento de voz o imágenes.

Este estilo se aplica en aquellos dominios de aplicación inmaduros, en particular BS se utiliza habitualmente para:

- Reconocimiento de voz.
- Reconocimiento de imágenes (detección de sospechosos, búsqueda de huellas dactilares, reconocimiento óptico de caracteres, etc.).
- Sistemas C4ISTAR (aplicaciones de índole militar pertenecientes al área de comando, control, inteligencia, vigilancia, adquisición de objetivos y reconocimiento).
- Toma de decisiones automáticas o semi-automáticas (otorgamiento de créditos, control vehicular, sistemas dinámicos complejos, planes de batalla, control de robots, etc.).
- Detección de anomalías a partir de grandes cantidades de información (intrusos, impurezas, semántica, etc.)

9.5.3 Componentes

En BS se utilizan tres tipos de componentes: **blackboard**, **fuente de conocimiento** y **sub-sistema de control**.

- Es un repositorio de datos dividido en **niveles**. Cada nivel almacena **soluciones parciales del problema que resuelve el sistema**; las soluciones parciales de un nivel son conceptualmente iguales entre sí y conceptualmente diferentes a las de otros niveles. Por ejemplo, si el sistema intenta convertir a texto un discurso, todas las sílabas que se detecten se almacenarán en el mismo nivel, en tanto que las palabras que se construyan a partir de aquellas se almacenarán en otro nivel. Al conjunto de todas las soluciones se lo llama espacio de soluciones.

  Los niveles están ordenados en relación a la **utilidad de sus soluciones**. Cuanto más incompleta o parcial es la solución más abajo se almacena; las soluciones finales se almacenan en el nivel

superior del Blackboard. Cada nivel del Blackboard, por consiguiente, puede tener una interfaz diferente. Sin embargo, para diseñar cada nivel se debería aplicar el POI por lo que todos los niveles deberían presentar una interfaz abstracta que oculte la forma en que se almacenan las soluciones.

Cada tipo de solución se define mediante un conjunto de atributos, y cada solución de ese tipo corresponde a un conjunto de pares (atributo, valor), es decir a la asignación de ciertos valores a los atributos del tipo correspondiente. Es importante tener en cuenta que normalmente es necesario mantener relaciones entre las soluciones almacenadas en los diferentes nivel. Sin embargo ningún nivel accede a los datos de otro nivel.

- **Fuente de Conocimiento.** Cada Fuente de Conocimiento (FC) implementa una regla o algoritmo que colabora para obtener una solución parcial o final; cada FC es un "experto" en resolver un aspecto del problema global. Ninguna FC puede resolver el problema por sí sola.

  Todas las FC tienen interfaces muy semejantes (en general son idénticas pero pueden diferenciarse en la forma que son parametrizadas o inicializadas). La regla o algoritmo de una FC está condicionada por una precondición. Lo usual es que cada FC presente una subrutina para determinar si su precondición se verifica en el momento de la invocación, y otra subrutina para disparar la regla o ejecutar el algoritmo. Esta última subrutina acepta un parámetro que corresponde al foco de atención.

  Otra alternativa es que cada FC comunique mediante un evento que su precondición se verifica, luego de lo cual otro componente podría invocar la ejecución de la regla o algoritmo que esta implementa.

  La regla o algoritmo de cada FC implementa una estrategia de razonamiento que puede ser hacia adelante (**forward reasoning**) o hacia atrás (**backward reasoning**). Una estrategia de razonamiento hacia adelante comienza por el conocimiento o datos básicos intentando arribar al objetivo del razonamiento; inversamente, una estrategia hacia atrás comienza por asumir el objetivo de la prueba y luego busca confirmar la existencia de los datos que hacen falta para probar esa conjetura.

  En términos del **Blackboard** una estrategia *forward reasoning* implica que:

  la **FC** *lee datos de un cierto conjunto de niveles y escribe su aporte en un nivel superior a todos ellos*;

  en tanto que una estrategia *backward reasoning* implica que:

  la **FC** *consulta ciertos niveles del Blackboard y hace su aporte escribiendo en un nivel inferior a todos ellos.*

  En ambos casos, los aportes pueden materializarse de dos formas o una combinación de ambas:

  - modificando algunos atributos de algunas soluciones parciales.
  - estableciendo relaciones entre las soluciones parciales de un mismo o diferentes niveles.

  Las relaciones así establecidas pueden tener uno de dos significados: **(i)** una solución parcial se construye o fundamenta en base a las soluciones parciales de niveles inferiores con las que está relacionada, o **(ii)** las soluciones parciales de un mismo nivel son partes de una misma solución parcial de un nivel superior que aun no ha sido deducida.

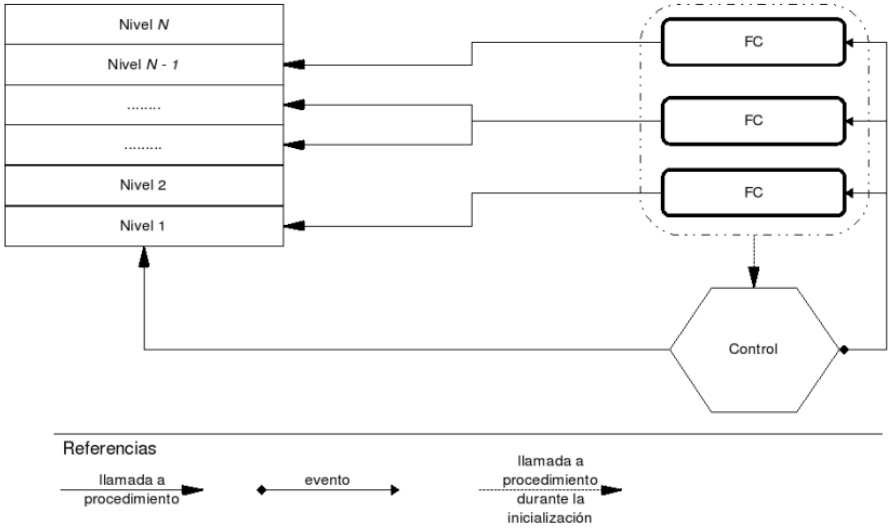
  Las FC, en conjunto, son responsables de establecer las relaciones entre una solución en un nivel y las soluciones en niveles inferiores que corroboran dicha solución.

- **Control.** Este subsistema es el encargado de coordinar la actividad de las **FC**. Lo usual en el estilo BS es que este componente implemente una estrategia de control o razonamiento *oportunista*, es decir, una estrategia de razonamiento que combina las otras dos estrategias, precisamente, de forma oportunista.

El **Control** monitorea los cambios en el **Blackboard** y decide qué FC debe ejecutarse, según la estrategia de control que implementa, pasándole el *foco de atención*. El foco de atención puede ser una **FC**, un conjunto de soluciones almacenadas en el **Blackboard** o una combinación de ambos. El **Control** es el responsable de determinar el foco de atención en cada momento.

La interfaz del **Control** suele ser una única subrutina que inicia el sistema pero también puede haber subrutinas para que las **FC** se registren y desregistren, y para que anuncien si su precondición se verifica.

Usualmente el **Control** se subdivide en dos o más componentes.



9.5.4 Conectores

El estilo **BS** no define nuevos tipos de conectores sino que utiliza eventos y llamada a procedimiento como se muestra en la Figura. Los eventos, en realidad, se implementan con una forma de invocación implícita ya que lo que se espera es que el Control no mantenga referencias estáticas a las **FC**.

9.5.5 Patrones Estructurales

El único patrón estructural admitido por el estilo **BS** se grafica en la Figura. Los diferentes niveles del **Blackboard** no se comunican entre sí aunque normalmente se deben mantener relaciones entre las distintas soluciones almacenadas en los niveles. Las **FC** pueden utilizar libremente la interfaz de cualquiera de los niveles del **Blackboard** pero no se comunican entre sí. El Control puede utilizar el **Blackboard** en modo de sólo-lectura, es decir sólo puede invocar los observadores de las interfaces de los niveles del **Blackboard**.

Las **FC** se registran ante el **Control** en el momento de la inicialización del sistema; por consiguiente el **Control** no tiene referencias estáticas a las **FC**. Al registrarse, cada **FC** comunica una referencia hacia las subrutinas que sirven para verificar su precondición y para ejecutar su acción. También puede ser necesario que cada **FC** comunique al **Control** ciertos *meta-datos* sobre su posible contribución a la solución del problema (por ejemplo, si implementa razonamiento hacia atrás o hacia adelante, de qué niveles lee información y en cuáles escribe, cuáles de los atributos de las soluciones modifica y/o consulta, etc.). El **Control** invoca dichas subrutinas según la estrategia de razonamiento que implementa. En la Figura esta interacción se grafica como un evento dado que el **Control** no tiene referencias estáticas a las interfaces de las **FC**.

9.5.6 Modelo Computacional Subyacente

El modelo computacional subyacente de un sistema basado en BS se expresa mediante el siguiente algoritmo:

1. Las **FC** se registran ante el **Control**, como se explica en la sección 9.5.5 (anterior).

2. El **Control** determina el foco de atención, consultando el **Blackboard** y las **FC** registradas.
3. El **Control** lista las **FC** cuya precondición se satisface. Si la lista es vacía el programa finaliza.
4. El **Control** analiza la contribución que cada **FC**, listada en el paso anterior, puede realizar.
5. En función del foco de atención y de las contribuciones que pueden realizar las **FC** en condiciones de aportar algo, el **Control** ejecuta una de ellas pasándole el foco de atención como parámetro.
6. La **FC** ejecutada realiza un cambio en uno o varios niveles del Blackboard, teniendo en cuenta el foco de atención.
7. El flujo de control retorna al **Control**.
8. El **Control** determina si se ha alcanzado la solución final o si se han agotado el tiempo disponible o los recursos del sistema como para continuar, en cualquiera de estos casos el programa finaliza.
9. El ciclo se repite desde el paso **2**.

Observar que el estilo estipula que la solución se construye de aun paso a la vez, desalentando, en consecuencia, la posibilidad de un modelo computacional concurrente o paralelo. Se considera que la posibilidad de concurrencia o paralelismo es muy compleja frente a estar intentando resolver un problema para el cual no hay un algoritmo claro y bien definido.

Como se mencionó en secciones anteriores el foco de atención puede ser **(a)** una FC, **(b)** un conjunto de soluciones del Blackboard o **(c)** una combinación de ambos. Claramente, si el foco de atención es **(a)** o **(c)**, en el paso **5**, el Control seleccionará la FC del foco. Si el foco de atención es **(c)** el Control dispone de todo como para acelerar el algoritmo de ejecución por lo que podría saltarse del paso **2** al paso **5**.

9.5.7 Invariantes Esenciales

Los invariantes esenciales del estilo son los siguientes:

- El **Blackboard** es pasivo respecto del resto de los componentes del sistema, es decir solo almacena datos.
- Las **FC** son los únicos componentes que pueden modificar el **Blackboard**. Las FC no pueden interactuar entre sí.
- Solo el **Control** ejecuta a las **FC**.
- El sistema debe construirse de forma tal que sea posible cambiar, eliminar y agregar **FC** (aunque no necesariamente de forma dinámica).
- No hay, necesariamente, una secuencia predeterminada para ejecutar las **FC**.

9.5.8 Metodología de Diseño

La metodología de diseño al utilizar **BS** se basa en resolver un problema en un dominio de aplicación inmnaduro para el cual aun no se conocen algoritmos deterministas eficientes.

1. **Definir el problema.**
    - Especificar el dominio del problema y las áreas generales de conocimiento para encontrar una solución.
    - Estudiar los datos de entrada para el sistema. Determinar las propiedades especiales de la entrada tales como ruido, variaciones, estabilidad, calidad, regularidad, etc.
    - Definir la salida del sistema, es decir las soluciones finales. Especificar los criterios de corrección y aceptación (es decir soluciones correctas y aceptables).
    - Detallar la interacción del usuario con el sistema. Tener en cuenta que la interacción del usuario puede ser una FC.

2. **Definir el espacio de soluciones para el problema.** Aquí podemos distinguir entre **soluciones totales** (se soluciona el problema por completo), **finales** (una parte del problema se soluciona por completo) y **parciales** (cualquier otra solución que no es ni total ni final). Por lo tanto se pueden seguir los siguientes pasos:

   - Especificar qué significa precisamente una solución final.
   - Listar los distintos niveles de solución para el problema.
   - Organizar las soluciones en una o más jerarquías respecto de la solución final (las soluciones más pobres estarán en la parte inferior de la jerarquía en tanto que las soluciones finales se ubicarán en la parte superior).
   - Buscar subdivisiones de las soluciones completas que pueden ser resueltas independientemente.

3. **Dividir el proceso para arribar a la solución en etapas.**

   - Definir cómo las soluciones de un nivel se transforman en soluciones de niveles superiores.
   - Describir cómo se establecerán hipótesis en un nivel.
   - Detallar cómo se corroborarán hipótesis en un nivel considerando datos en niveles inferiores.
   - Especificar la clase de conocimiento que puede usarse para excluir partes del espacio de soluciones. Por ejemplo, el cielo puede excluirse en la detección de koalas ya que estos animalitos son muy simpáticos pero no vuelan.

4. **Dividir el conocimiento en fuentes de conocimiento especializadas.** Se debe garantizar que existan las fuentes de conocimiento necesarias como para que para la mayoría de las entradas que reciba el sistema exista una secuencia de invocación que lleve a una solución aceptable.

5. **Refinar el espacio de soluciones y diseñar el Blackboard**. Refinar la primera definición del espacio de soluciones dada en el paso **2** para ir definiendo los niveles del **Blackboard** y la interfaz de cada uno de ellos. La interfaz de cada nivel se debe seleccionar de forma tal que sea posible agregar nuevas **FC** y modificar las existentes sin incurrir en grandes costos. Al mismo tiempo se debe ir determinando los atributos que describen las soluciones en cada nivel.

6. **Diseñar y especificar el sub-sistema de control.** El diseño de una buena estrategia de control es la parte más compleja de un sistema basado en BS. Usualmente implica un proceso tedioso de prueba y error combinando varios mecanismos y estrategias parciales. El patrón de diseño **Strategy** puede ser muy útil en esta parte del diseño.

   También es recomendable usar el patrón **Command** para registrar las **FC** ante el **Control**.

   Considerar en subdividir el **Control** en dos o más componentes. Por ejemplo, es muy común tener un componente dedicado a mantener datos de control tales como el foco de atención, los meta-datos de las **FC** registradas, etc. También es posible que la estrategia de control pueda ser lógicamente particionada lo que suele ser conveniente de reflejar en la descomposición del **Control**.

7. **Diseñar las FC**. Asignar cada una de las fuentes de conocimiento definidas en el paso **4** a un componente tipo **FC**. Tener en cuenta que las **FC** deben ser independientes entre sí y del **Control** (en lo referente a la estrategia que este aplica aunque no respecto de su existencia o de los meta-datos que este espera). Cada **FC** debe estructurarse al menos en partes-condición y partes-acción, pero tener en cuenta que algunas **FC** pueden ser programas bastante complejos por lo que sería conveniente diseñarlos con cuidado.

9.5.9 Ventajas y Desventajas

El estilo BS tiene las siguientes ventajas en relación a su ámbito de aplicación:

- Alienta la experimentación en dominios de aplicación en los cuales no existen soluciones algorítmicas claras y bien definidas.

- La independencia mutua de las FC permite modificar fácilmente el sistema eliminando, modificando o agregando fuentes de
- conocimiento. Al separar la estrategia de control en un componente específico es posible alterar el funcionamiento global del sistema con un único cambio.
- El sistema siempre provee una solución aunque sea parcial, aproximada y débilmente corroborada.

Sin embargo, las soluciones dentro de este estilo sufren de las siguientes desventajas:

- Se torna dificultoso el testing dado que el sistema no funciona, necesariamente, según un algoritmo determinista.
- No hay garantía de obtener una solución final correcta. Es complicado establecer la estrategia de control; es necesario experimentar con diferentes estrategias.
- Suelen ser sistemas con un desempeño pobre debido a la falta de un algoritmo determinista (sin embargo esto es el peor de dos males cuando no se dispone de tal algoritmo).
- Requieren un largo período de desarrollo debido a los varios experimentos que hay que realizar hasta dar con un conjunto de **FC** y una estrategia de razonamiento adecuada que garantice buenas soluciones en la mayoría de los casos.
- No soporta de forma directa y evidente una implementación concurrente o paralela. La mera existencia del **Blackboard** como repositorio central torna muy complicado implementar un algoritmo concurrente.

9.5.10 Documentación

- **Diagrama Canónico.** Se debe documentar el diagrama canónico como se muestra en la Figura. Es decir, se deben documentar los niveles del **Blackboard** asignándoles un nombre significativo o una designación muy breve, cada una de las **FC** mostrando con qué niveles del **Blackboard** interactúan y el sub-sistema de control con los módulos que lo componen.
- **Guía de Módulos.** Se debe describir la función de cada **FC**, su aporte, y todos los metadatos que la definen. También se deben describir los niveles del **Blackboard** con sus interfaces, soluciones que almacenan, atributos que definen las soluciones en cada nivel y la forma de establecer relaciones entre las soluciones de los distintos niveles. Finalmente, se deben describir los componentes del **Control**, especificar la estrategia de control, etc.
- **Estructura de Módulos.** Se divide naturalmente en tres módulos lógicos: **Blackboard**, **FC** y **Control**. A su vez el módulo **Blackboard** se subdivide en un módulo **por cada nivel**; en tanto que **FC** se subdivide en **un módulo por FC**. **Control** también puede subdividirse. Los módulos del segundo nivel pueden, a su vez, subdividirse se siguen siendo complejos.
- **Conector Control-FC.** Se debe documentar precisamente la forma en que interactuarán las **FC** con el **Control** respetando las premisas estipuladas en las secciones anteriores.

9.5.11 Especializaciones Comunes

No se han documentado especializaciones de este estilo.

9.5.12 Deformaciones Comunes

Una deformación más o menos común es que el **Control** almacene los datos de control (foco de atención, por ejemplo) en el **Blackboard**.

Otra posibilidad es que el **Blackboard** tenga dos o más paneles.

9.6 Cliente/Servidor de Tres Capas

También conocido como *Cliente/Servidor de n capas, Arquitectura en capas y Sistemas distribuidos* (aunque es un nombre demasiado genérico)

9.6.1 Propósito

Descomponer el procesamiento y almacenamiento de los datos procesados por grandes sistemas corporativos, de forma tal que se verifiquen las siguientes cualidades:

- **Integración de datos y aplicaciones.** Unidades corporativas diferentes que desarrollaron sus propios sistemas de procesamiento de datos deben unificar esos sistemas; empresas diferentes con diferentes sistemas se fusionan y por lo tanto lo mismo debe ocurrir con sus sistemas.
- **Modificabilidad** de aplicaciones, de representación de los datos, de ubicación física de los componentes.

  Las reglas del negocio cambian constantemente lo que implica cambios en las aplicaciones y en la representación de los datos; la organización crece y su ubicación física se expande por lo que es necesario que las aplicaciones y los datos migren.
- **Escalabilidad** para permitir que el sistema acompañe el crecimiento de la organización de forma transparente para las unidades que ya están en producción.
- **Aumentar el desempeño** para que más usuarios puedan utilizar el sistema.

Todo esto en un contexto de grandes cantidades de datos y transacciones. Otras cualidades que se buscan son: tolerancia a fallas, continuidad de las operaciones, alta redundancia y seguridad.

9.6.2 Aplicabilidad

Este estilo se aplica casi exclusivamente a grandes sistemas de gestión corporativa llamados ERP (*Enterprise Resource Planning*). Estos sistemas son utilizados por lo general por grandes corporaciones pero el estilo también es útil para diseñar los hermanos menores de estos sistemas utilizados por PyMES. También se utiliza mucho para conectar o acceder este tipo de sistemas desde Internet o para estructurar sistemas de comercio o gobierno electrónico (o sistemas que se basan fuertemente en Internet).

9.6.3 Componentes

Existen dos tipos de componentes: **clientes** y **servidores**. Los clientes solicitan servicios a otros componentes; los servidores proveen servicios, que van desde subrutinas a bases de datos completas, a otros componentes. Es común que un servidor sea cliente de otro servidor.

Clientes y servidores suelen agruparse en **capas**, también llamadas capas lógicas, cada una de las cuales, muy posiblemente, ejecuta en una plataforma diferente. Cada capa ofrece o solicita un conjunto de servicios y datos (que es la unión de los servicios y datos que ofrecen los servidores dentro de esa capa o los servicios y datos que solicitan los clientes en esa capa).

Normalmente una capa es un gran sistema de software (que incluye aplicaciones y datos) por lo que debe ser descompuesto. Cada componente de una capa puede ser un sistema, sub-sistema, un TAD o un módulo. Usualmente se utilizan tres capas.

Las capas NO deben confundirse con los estratos de los Sistemas Estratificados. Las capas no suelen diseñarse pensando en máquinas abstractas sino que el criterio se basa en escalabilidad, desempeño, tolerancia a fallas, uso inteligente del ancho de banda, etc

En un sistema **CS3C** los servidores pueden cumplir funciones muy diversas. Podemos clasificarlos en dos categorías: de **negocio** y de **infraestructura**, también llamados de **soporte**. Dentro de la primera categoría se encuentran las aplicaciones que implementan un requisito funcional específico del negocio; en la segunda categoría están los servidores que cumplen funciones más generales que usualmente abarcan más de un dominio de aplicación.

9.6.4 Conectores

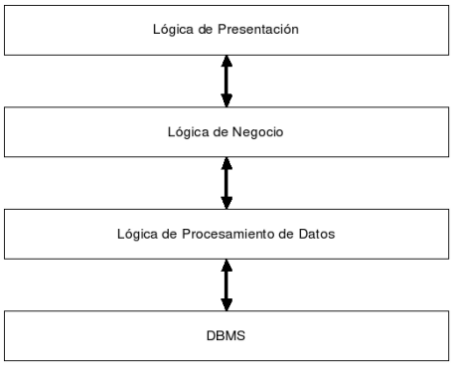
- **Protocolos cliente/servidor.** Por ejemplo: FTP, HTTP, SQL remoto, RPC, NFS, two-phase commit protocols, Distributed Transaction Processing (DTP) de IBM, Webservices, etc.
- Llamada a procedimiento.

- Llamada a procedimiento remota (RPC, RMI, etc.).
- Tubos.
- Memoria compartida (aunque debería utilizarse sólo en casos muy especiales).

Usualmente la comunicación entre componentes es de a pares y la inicia un cliente; al pedido de un servicio de un cliente le corresponde la respuesta del servidor respectivo. En otras palabras la comunicación entre cliente y servidor es, por lo general, asimétrica.

Algunos protocolos deben ser capaces de proveer integridad de las transacciones que llevan a cabo cooperativamente clientes y servidores.

Otras propiedades de los conectores pueden ser: estrategia para el manejo de errores, cómo se inicia y termina una interacción entre un cliente y un servidor, existencia de sesiones, estrategia para la localización de los servidores, etc.



Cada rectángulo representa una capa lógica; las flechas indican comunicación bidireccional por medio de diversos conectores.

9.6.5 Patrones Estructurales

Los patrones estructurales indican las formas en que se pueden distribuir las aplicaciones (clientes y servidores) y los datos en capas y las formas en que todos estos elementos pueden interactuar entre sí. La Figura muestra las relaciones estructurales básicas.

- **Las capas físicas.** Los patrones estructurales están guiados por la topología física de la red de la corporación. Todos sus componentes físicos están conectados a una o varias redes con un ancho de banda de medio a alto.

  Estos componentes están organizados en tres capas físicas: PCs, servidores medianos y mainframes. En organizaciones muy grandes estas tres capas pueden llegar a ser cuatro o cinco; por lo general es la capa de servidores medianos la que se subdivide.

  Utilizaremos el término cliente físico para referirnos tanto a las PCs como a los servidores medianos, y servidor físico para referirnos a los servidores medianos y a los mainframes.

  Si bien las capas físicas no son parte de lo que el ingeniero de software debe diseñar es importante mencionarlas porque como ya dijimos es una de las guías o criterios para seleccionar las capas lógicas.

- **Las capas lógicas.** Un sistema típico en este estilo consiste de cuatro capas lógicas (como se ve en la Figura)

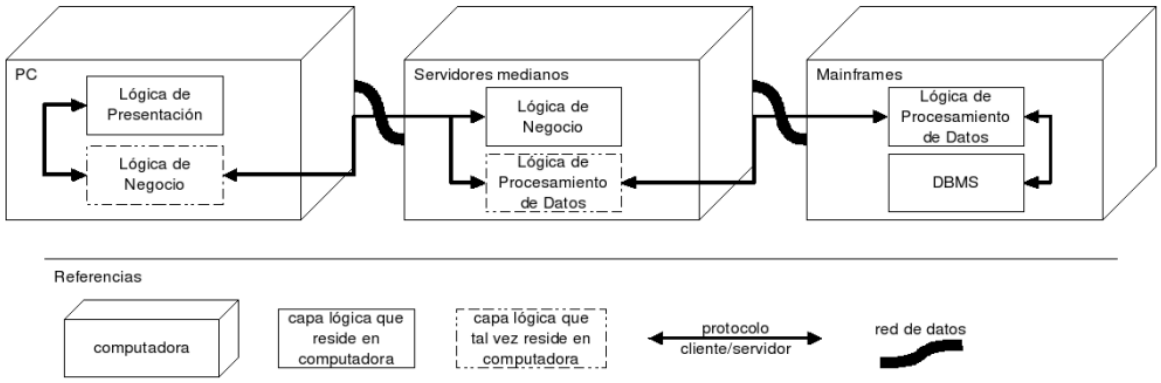
  - **Lógica de Presentación (LP).** Esta es la parte del código que interactúa con un dispositivo como una PC o terminal de autoservicio. Esta capa se encarga de cosas como disposición de los elementos gráficos en la pantalla, escribir los datos en pantalla, manejo de ventanas, manejo de los eventos del teclado y mouse, etc.

- **Lógica de Negocio (LN).** En esta capa se codifican las reglas del negocio. Por ejemplo, si el sistema es de un banco en esta capa se programan conceptos como plazo fijo, cuenta corriente, cheque, etc.
- **Lógica del Procesamiento de los Datos (LPD).** En esta capa se oculta la forma en que se consultan o almacenan los datos persistentes; en general es uno de los dialectos de SQL. Aunque esto debería estar oculto para la mayoría de los componentes de la Lógica de Negocio usualmente no lo está, lo que provoca importantes costos de mantenimiento y cambio. Por lo general, cualquier componente en de la Lógica de Negocio puede utilizar SQL. El problema es que esta interfaz (SQL) no encapsula lo que se denomina estructura física de los datos, es decir la representación en tablas relacionales de los datos. Por lo tanto, un cambio en esa estructura física suele tener un gran impacto en la Lógica de Negocio.
- **DBMS.** Esta capa usualmente está formada por uno o varios **RDBMS** pero no es extraño encontrar otros componentes como el sistema de archivos de sistemas operativos como UNIX, AS/400, Windows, etc. Si esto ocurre, entonces al SQL de la capa anterior se le suman las rutinas para acceder al sistema de archivos local o remotamente.

En general, la **LPD** accede los datos almacenados en un **DBMS**, la **LN** los procesa y la **LP** muestra los resultados al usuario. Las capas pueden interactuar entre sí de la siguiente forma: la **LP** es cliente de la **LN**; la LN es servidor para la **LP** y cliente para la **LPD**; la **LPD** es servidor para la **LN** y cliente para el **DBMS**; el **DBMS** actúa únicamente como servidor de la **LPD**. Lograr preservar estas interacciones requiere de una ingeniería muy cuidadosa del sistema.

El término procesamiento de la aplicación refiere a la unión de las capas **LN** y **LPD**.

El problema consiste, entonces, en determinar cómo se distribuyen estas capas lógicas en las capas físicas.



- **La forma más común de distribución.** La forma más usual de distribuir tanto datos como aplicaciones se muestra en la Figura.

  Esto significa que por lo general la LP ejecuta sobre las PCs; posiblemente algo de la LN ejecute también sobre las PCs. La LN ejecuta normalmente sobre los servidores medios aunque también puede ocurrir que estos también contengan algo de la LPD, por ejemplo un middleware para ruteo de transacciones, y no es extraño que haya algún DBMS. Por lo general, los DBMS yacen en los grandes servidores corporativos.

  En los tres puntos que siguen se analizarán más alternativas y se estudiarán sus ventajas y desventajas relativas.

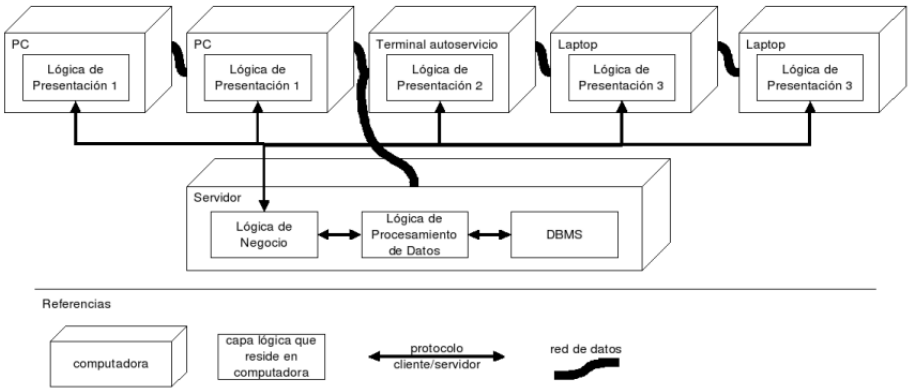
- **Distribución de la presentación.** La **LP** se distribuye en **dos** sentidos:

  1. Se la separa de las otras funcionalidades del sistema (LN, LPD y DBMS). El objetivo es desacoplar la presentación de los resultados del sistema y la entrada de datos de las reglas

de negocio y de la forma en que los datos se almacenan persistentemente. De esta forma es posible modificar la LP sin tener que hacer grandes modificaciones a las restantes capas. Además, al poner la LP sobre las PCs se hace un uso mucho más eficiente del poder de cómputo de estos equipos.

2. Puede haber diferentes programas que la implementen en diferentes grupos de PCs.

   Esto puede ocurrir debido a que haya otros tipos de computadoras más allá de las PCs (como terminales de autoservicio, terminales de texto, clientes finos, etc.) y debido a que diferentes PCs utilizarán diferente aplicaciones que requieren interacciones distintas.



- **Distribución del procesamiento de la aplicación.** Recordemos que el término *procesamiento de la aplicación* refiere a la **unión de las capas LN y LPD**. Por lo tanto, en esta sección analizaremos las tres alternativas para distribuir la LN y la LPD.

  El procesamiento de la aplicación reside únicamente en los clientes físicos. Esto significa que tanto los programas que implementan la LN como los que implementan la LPD están almacenados y ejecutan en los clientes físicos, es decir en las PCs o servidores medios.

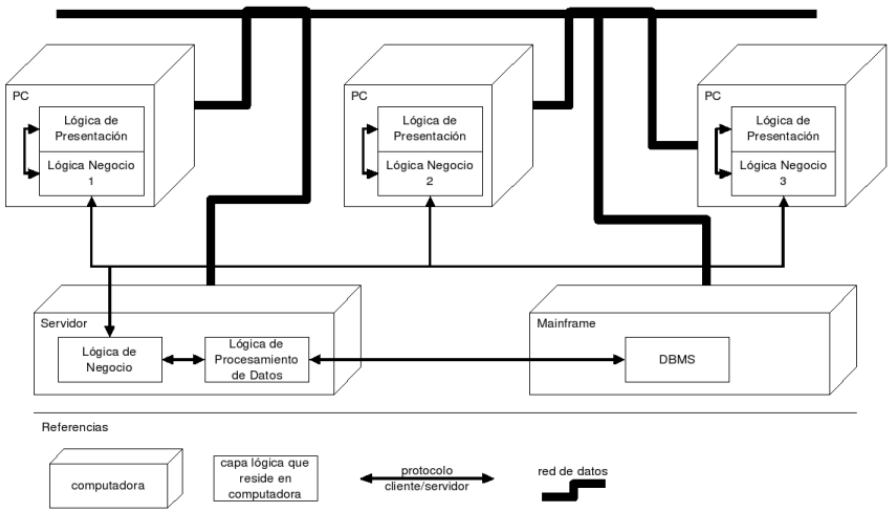
  En esta alternativa lo usual es que una porción de la LN esté en las PCs y el resto más la LPD en los servidores medianos. La porción de la LN que corre en las PCs es la más cercana a la interacción con el usuario.

  Entre las ventajas de estructurar el sistema de esta forma tenemos:

  - Es conveniente liberar recursos de los mainframes trasladando parte del procesamiento a los servidores medianos y las PCs debido al más bajo costo relativo de estos en relación con los grandes servidores.
  - Dado que ciertas reglas de negocio tienen mucho que ver con las operaciones de entrada/salida y poco con los datos persistentes (por ejemplo que un DNI tenga 8 dígitos o que una fecha esté en el calendario), resulta conveniente alojar esta porción de la LN lo más cerca posible de la LP para reducir el tráfico en la red.
  - Se reduce el tráfico en la red debido a la cercanía entre la LP y el procesamiento de la aplicación.

  Sin embargo, tenemos las siguientes deficiencias:

  - Se torna muy compleja la administración y mantenimiento de múltiples copias de la LN que residen en diferentes máquinas.
  - Es posible que las estaciones de trabajo no puedan llevar a cabo todo el procesamiento que se les pide con un nivel de desempeño aceptable (considérese el caso de una PC normal que debe correr aplicaciones Java).
  - Hay un esfuerzo por lograr un acceso sincronizado a los datos.



El procesamiento de la aplicación reside únicamente en los servidores físicos. En este caso los programas que implementan la LN y la LPD residen entre los servidores intermedios y los mainframes. En este caso, normalmente, la LN reside en los servidores en tanto que la LPD corre en los mainframes junto al DBMS. Esto significa que si un componente de la LN necesita acceder datos persistentes vía SQL deberá enviar la petición SQL a través de la red, la cual será atendida por un programa que ejecuta en los servidores centrales. Otra alternativa es poner parte de la LN en los servidores centrales.

Las ventajas de este patrón estructural son:

- Se elimina la redundancia de código, se simplifica la administración de los programas y se utilizan al máximo los servidores intermedios.
- Si parte de la LN yace en los mainframes entonces se reduce el tráfico en la red debido a las consultas sobre el DBMS.
- Se reduce notablemente el problema de sincronizar distintas aplicaciones que acceden a los mismos datos.

Pero tenemos estas desventajas:

- Puede ocurrir que con el tiempo se agoten los recursos de los servidores intermedios.
- Se incrementa significativamente el tráfico en la red entre la LP y la LN y puede haber tardanzas intolerables para los usuarios finales al momento del ingreso de datos.
- Se tiende a sub-utilizar el poder de cómputo de las PCs lo que implica una reducción en el retorno de la inversión en este hardware.

El procesamiento de la aplicación está distribuido entre los clientes y los servidores físicos. Esta alternativa puede combinar las ventajas de las dos anteriores y al mismo tiempo eliminar sus desventajas. Sin embargo es la más complicada desde el punto de vista de la ingeniería del sistema. Requiere una planificación sumamente cuidadosa, un diseño prolijo y una implementación acorde.

Por lo general en esta alternativa, la parte de la LN más relacionada con la entrada/salida se ubica en las estaciones de trabajo, el resto de la LN en los servidores intermedios y la LPD en los mainframes. En consecuencia se utilizan al máximo todos los equipos, se reduce el tráfico entre LP y LN a la transferencia de datos persistentes y se mantiene en un nivel aceptable la complejidad de la administración de múltiples copias del código.

- **Distribución de datos.** Es el turno de analizar los diferentes esquemas para la distribución de los datos. Obviamente hay una relación muy estrecha entre ambos en tanto que los primeros generan y necesitan de los segundos.

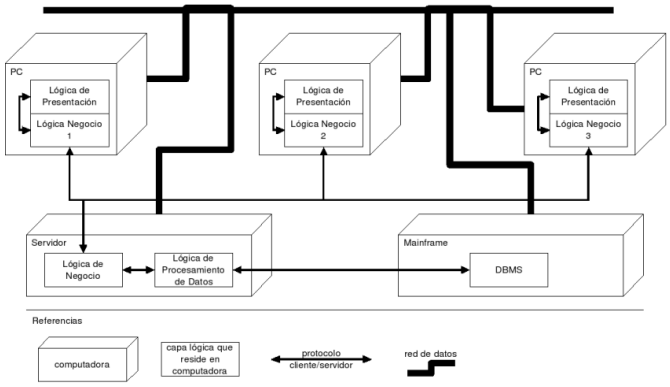
En el primer caso, llamado conexión directa todos los datos se concentran en los servidores centrales en tanto que parte de la LPD los accede a petición de la LN. Si bien es un esquema en principio correcto, hay casos donde es poco práctico y otros donde es imposible.

En el segundo caso, llamado conexión indirecta, existen dos tipos de DBMS, uno llamado local porque reside en un cliente o servidor intermedio y el otro llamado corporativo que reside en un mainframe. En el DBMS local se almacenan datos relativos a ese cliente (por ejemplo las ventas de un vendedor se almacenan en su laptop) o a ese servidor intermedio (por ejemplo los datos de las cuentas de los clientes de cierta sucursal de un banco); en tanto que en el servidor corporativo se almacenan los datos globales de la organización y también, cada cierto tiempo, este sincroniza con todos los servidores locales para tener una imagen única de los datos. Los servidores locales suelen ser DBMS de porte chico (por ejemplo MySQL o Microsoft SQL Server) mientras que los servidores corporativos son por lo general productos de gran porte (como Oracle, Informix, etc.).

La ventaja del esquema indirecto es que se reduce el tiempo de acceso a ciertos datos y el tráfico en la red. Al mismo tiempo, esta organización torna más complejo mantener la integridad de la base de datos global.

Una tercera alternativa es que los servidores corporativos sean más de uno por lo que LDP debería saber dónde buscar cada conjunto de datos específico. Aquí surge el mismo problema que cuando se divide la LN en varios servidores (cf. al subestilo broker), por lo que se aplica la misma solución: *brokers* de datos, también llamados *ruteadores de transacciones* o *middleware para transacciones*.

Claramente los patrones estructurales correspondientes a la distribución del procesamiento de la aplicación pueden combinarse con los patrones que surgen de los distintos esquemas de distribución de datos.



9.6.6 Modelo Computacional Subyacente

Los clientes inician transacciones o pedidos de datos a los servidores. Los servidores pueden propagar estos pedidos a otros servidores. Finalmente, los servidores retornan datos a los clientes. Clientes y servidores pueden ejecutar concurrentemente tanto de forma sincrónica como asincrónica.

9.6.7 Invariantes Esenciales

Los invariantes deben imponer una estructura en capas:

- La LP actúa siempre como cliente y solicita servicios sólo a la LN
- La LN solicita servicios sólo a la LPD
- La LPD solicita servicios sólo al DBMS
- El DBMS actúa únicamente como servidor
- Los clientes deben iniciar todas las transacciones
- Los servidores no tienen por qué conocer la identidad de los clientes antes de que estos soliciten un servicio

9.6.8 Metodología de Diseño

Indicaremos los pasos principales de una metodología más o menos neutral, general e ideal. Son los siguientes:

1. **Reingeniería.** Hoy día prácticamente no existe una organización que no cuente con un sistema tipo ERP. En la mayoría de los casos no existe documentación actualizada y es económicamente inviable desarrollar un nuevo ERP desde cero. Por lo tanto, el diseño de un nuevo ERP o la modernización del existente usualmente comienza con un costoso y (en el mejor de los casos) largo proceso de reingeniería con el objetivo de reutilizar al máximo posible el código y los datos (o mejor dicho la organización de los datos) existentes. Existen algunas herramientas comerciales que asisten en este proceso. El resultado de este paso debería ser la documentación del viejo ERP.

   En el caso de aplicaciones para alguna forma de comercio electrónico esta fase no será tan compleja pero seguramente este sistema deberá interactuar con sistemas existentes por lo que siempre es necesario comprenderlos antes de poder comenzar.

2. **Tecnología.** Si bien desde el punto de vista académico no es lo más recomendable pensar una arquitectura en términos de las tecnologías a utilizar, la realidad y la complejidad de estos sistemas torna inevitable y deseable la compra de ciertos productos que acorten los tiempos de desarrollo. Estos productos poseen ciertas interfaces que habilitan o desalientan ciertas interacciones. El desconocimiento de estas restricciones ha llevado a más de un proyecto al fracaso.

   Otro punto importante es que estas tecnologías suelen estar inmaduras al momento de querer aplicarlas y suele haber muy pocos desarrolladores que las dominen. Por lo tanto, se requiere de un período no menor de entrenamiento y pruebas hasta lograr que el equipo de desarrollo pueda producir con una productividad razonable.

3. **Distribución.** La tecnología con la que se trabajará más los requerimientos funcionales y no-funcionales del sistema guían o determinan los posibles esquemas de distribución de datos y aplicaciones.

   - Se deben analizar todas las alternativas presentadas en la sección 9.6.5.
   - Se debe seleccionar la más adecuada.
   - Se la debe documentar con precisión.
   - Se deben documentar las razones por las cuales se la seleccionó y no se seleccionaron las otras.

4. **Diseño de la Lógica de Presentación.** Por lo general, en esta fase hay dos grandes variantes:

   - **Desarrollo de un cliente.** No es la tendencia más actual pero no se la debe descartar de plano. No es tan complejo como parece si se utilizan las bibliotecas estándar o APIs para manejo de interfaces de usuario provistas por diversos sistemas o lenguajes de programación.
   - **Uso de un cliente estándar.** El caso típico es utilizar un navegador Web más programación en lenguajes como Javascript, HTML, etc. Lo cierto es que en definitiva un cliente de estas características provee de las cuestiones más básicas de la LP. Queda como tarea para el equipo de desarrollo definir el look-and-feel de la aplicación, los menús, formularios, ventas de error, etc. Es decir debe hacerse gran parte del trabajo que se hace en la otra alternativa. La gran ventaja está en que cualquier cliente físico tiene uno de estos clientes por lo que la organización no debe proveer uno (ventaja inestimable para aplicaciones Web).
   - **Diseño de la Lógica de Negocios.** Esta es tal vez la parte más compleja pero a la vez para la que se dispone de más herramientas metodológicas. Para el diseño de parte de esta capa usualmente se utiliza un Diseño Orientado a Objetos el cual se implementa en un lenguaje orientado a objetos como Java, C#, etc.

     La otra parte corresponde al código existente el cual carece de un diseño claro (usualmente es alguna forma de diseño funcional o estructurado) y está implementado en lenguajes como COBOL, 4GL, RPG, etc.

     Al menos en la parte de la LN que debe ser implementada desde cero se deberían aplicar todas las consideraciones del DBOI.

- **Diseño de la Lógica de Procesamiento de Datos.** Usualmente no queda mucho por hacer en esta capa porque por lo general los productos y la tecnología utilizados proveen todo lo necesario. Como ya mencionamos la LPD suele estar formada por una o más versiones de SQL (que corresponden a uno o más RDBMS) y por las rutinas de acceso a sistemas de archivos locales o remotos. Cuando existen múltiples repositorios de datos no es conveniente que las otras capas lo sepan y al mismo tiempo se torna complejo dirigir y manejar apropiadamente las transacciones por lo que se suele incorporar un middleware para ruteo, encolamiento, sincronización y procesamiento de transacciones.

9.6.9 Ventajas y Desventajas

Correctamente aplicado al tipo de sistemas para el cual fue pensado este estilo no presenta desventajas importantes y al mismo tiempo permite alcanzar ciertas cualidades muy importantes (ver sección 9.6.1).

Su principal desventaja radica en que los sistemas para los que sirve son muy complejos, el entorno donde se realiza el desarrollo está plagado de intereses políticos y presiones comerciales y el estilo en sí tiene muchas variantes y cuestiones que deben ser decididas por el arquitecto

9.6.10 Documentación

- **Distribución de aplicaciones.** Es fundamental documentar detalladamente la distribución de aplicaciones. Usualmente se lo hace a través de la estructura física.

- **Distribución de datos.** Es fundamental documentar detalladamente la distribución de datos. Usualmente se lo hace a través de la estructura física.

- **Estructura de Módulos.** La Estructura de Módulos de un sistema basado en CS3C debe seguir la estructura de capas del estilo. Es muy importante documentar la pertenencia de cada módulo del diseño a una de las capas junto a una justificación que fundamente la necesidad de que ese módulo esté en esa capa. Esta documentación fuerza a los ingenieros a justificar cada decisión arquitectónica reduciendo las posibilidades de que un módulo sea incluido en una capa a la que no debería pertenecer.

- **Protocolos.** Deben documentarse los protocolos que se utilizarán en cada interacción entre clientes y servidores. Usualmente esto es parte de la tecnología que se compra a terceros3. La mayoría de las implementaciones de los protocolos corresponden a estándares pero cada implementación es diferente. Deben documentarse las diferencias respecto del estándar.

- **Tecnología e infraestructura.** Como ya mencionamos, en el desarrollo y despliegue de estos sistemas se utiliza gran cantidad de aplicaciones compradas a terceros. Es importante inventariar estos productos determinando su función, relaciones, ubicación, distribución, etc.

9.6.11 Especializaciones Comunes

- Las tres capas físicas pueden ampliarse a un mayor número lo que aumenta el número de alternativas de distribución del procesamiento y de los datos. En general aplican consideraciones similares a las oportunamente consignadas en la sección 9.6.5.

- Los servidores pueden notificar a los clientes de ciertas situaciones sin que haya mediado un pedido de servicio por parte de estos.

- Normalmente se preserva la propiedad de que el servidor no conoce la identidad de los clientes utilizando eventos o callbacks.

- Introducción dinámica de clientes y servidores. Limitaciones en el número de clientes o conexiones que un servidor puede manejar simultáneamente.

9.6.12 Deformaciones Comunes

Son innumerables y en general corresponden a errores en el diseño o a conseciones en favor de mejor desempeño, tolerancia a fallas, presiones del mercado o de los usuarios, etc.

Debe tenerse en cuenta que este estilo es en sí mismo una especialización del estilo Cliente/Servidor más general, es decir el cual no impone una restricción estructural en capas. Por lo tanto, ciertas deformaciones del estilo aquí consignado son, en realidad, sistemas basados en el estilo más general.

Otra deformación de este estilo, que a la vez es un estilo en sí mismo y también una deformación del estilo general Cliente/Servidor, es lo que se conoce como sistemas *Peer to Peer*(P2P). En este estilo la asimetría entre clientes y serividores se pierde por lo que todos los componentes se convierten en pares. En este sentido, en principio, cualquier componente puede interactuar con cualquier otro solicitando sus servicios. Por lo tanto, los conectores de este estilo pueden involucrar complejos protocolos bidireccionales.

10 Testing

10.1 Introducción

El testing de software pertenece a una actividad o etapa del proceso de producción desoftware denominada Verificación y Validación, usualmente abreviada como V&V.

V&V es el nombre genérico dado a las actividades de comprobación que aseguran que el software respeta su especificación y satisface las necesidades de sus usuarios. El sistema debe ser verificado y validado en cada etapa del proceso de desarrollo utilizando los documentos (descripciones) producidas durante las etapas anteriores.

10.1.1 Definición de testing y vocabulario básico

El **testing** es una actividad desarrollada para evaluar la calidad del producto, y para mejorarlo al identificar defectos y problemas. El testing de software consiste en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados a partir del dominio de ejecución que usualmente es infinito, en relación con el **comportamiento esperado**.

Testear un programa significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados. El testing se estructura en casos de prueba o casos de test; los casos de prueba se reúnen en conjuntos de prueba. Desde el punto de vista del testing se ve a un programa (o subrutina) como una función que va del producto cartesiano de sus entradas en el producto cartesiano de sus salidas. Es decir:

$$P : ID \rightarrow OD$$

donde *ID* se llama dominio de entrada del programa y *OD* es el dominio de salida. Normalmente los dominios de entrada y salida son conjuntos de tuplas tipadas cuyos campos identifican a cada una de las variables de entrada o salida del programa, es decir:

$$ID = [x_1 : X_1, ..., x_n : X_n]$$
$$OD = [y_1 : Y_1, ..., y_m : Y_m]$$

De esta forma, un caso de prueba es un elemento, x, del dominio de entrada (es decir $x \in ID$) y testear P con x es simplemente calcular $P(x)$. En el mismo sentido, un conjunto de prueba, por ejemplo T, es un conjunto de casos de prueba definido por extensión y testear P con T es calcular $P(x)$ para cada $x \in T$. Es muy importante notar que x es un valor constante, no una variable.

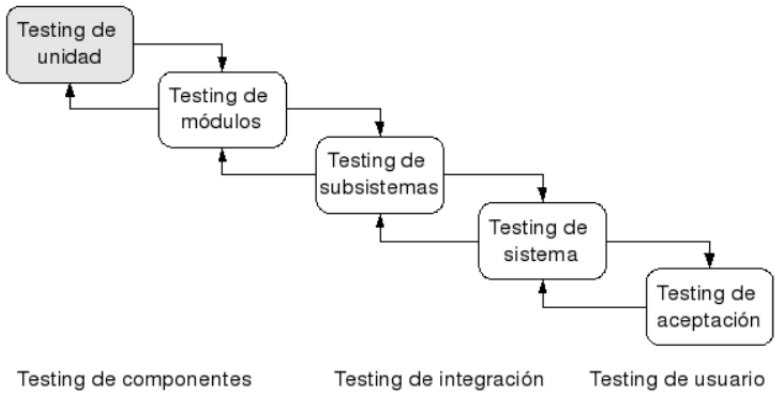
También es importante remarcar que $x_1, ..., x_n$ son las entradas con que se programó el programa (esto incluye archivos, parámetros recibidos, datos leídos desde el entorno, etc.) y no entradas abstractas o generales que no están representadas explícitamente en el código. De misma forma, $y_1, ..., y_m$ son las salidas explícitas o implícitas del programa (esto incluye salidas por cualquier dispositivo, parámetro o valor de retorno, e incluso errores tales como no terminación, *Segmentation fault*, etc.).

De acuerdo a la definición clásica de corrección, un programa es correcto si verifica su especificación. Entonces, al considerarse como técnica de verificación el testing, un programa es correcto si ninguno de los casos de prueba seleccionados detecta un error. Precisamente, la presencia de un error o defecto se demuestra cuando $P(x)$ no satisface la especificación para algún x en ID.

10.1.2 El proceso de Testing

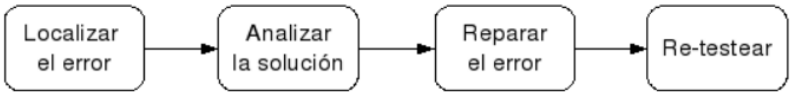
Es casi imposible, excepto para los programas más pequeños, testear un software como si fuera una única entidad monolítica. Como hemos visto en los capítulos sobre arquitectura y diseño, los grandes sistemas de software están compuestos de subsistemas, módulos y subrutinas.

Como sugiere la siguiente Figura,



un sistema complejo suele testearse en varias etapas que por lo general se ejecutan siguiendo una estrategia bottom-up, aunque el proceso general es iterativo. Se debe volver a las fases anteriores cada vez que se encuentra un error en la fase que está siendo ejecutada.

En general, una vez detectado un error se sigue el proceso graficado en la siguiente Figura.



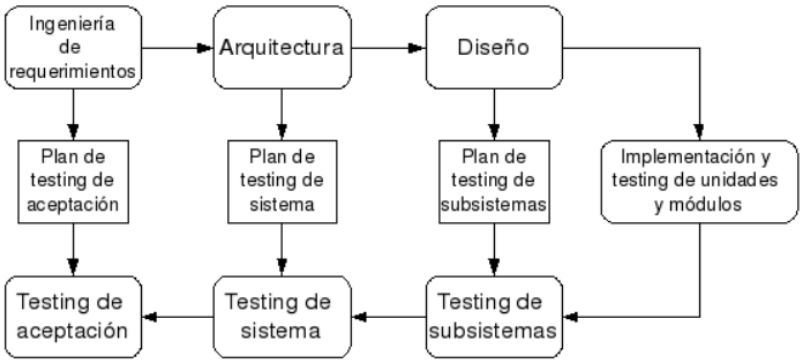
De aquí que a las iteraciones del proceso de la Figura 1 se las llame *re-testing*.

Dado que no hay una definición precisa de subsistema e incluso de unidad, el proceso de testing sugerido debe ser considerado como una guía que debe ser adaptada a cada caso específico.

En la práctica industrial usualmente se entiende que el testing es una actividad que se realiza una vez que los programadores han terminado de codificar; en general es sinónimo de testing de aceptación. Entendido de esta forma el testing se convierte en una actividad costosa e ineficiente desde varios puntos de vista:

- Los testers estarán ociosos durante la mayor parte del proyecto y estarán sobrecargados de trabajo cuando este esté por finalizar.
- Los errores tienden a ser detectados muy tarde.
- Se descubre un gran número de errores cuando el presupuesto se está terminando.
- Los errores tienden a ser detectados por los usuarios y no por el personal de desarrollo –lo que implica un desprestigio para el grupo de desarrollo.

Por lo tanto, se sugiere un proceso de testing mejor imbricado con el proceso general de desarrollo, como se muestra en la siguiente Figura:



10.1.3 Las dos metodologías clásicas de testing

Tradicionalmente el testing de software se ha dividido en dos estrategias básicas que se supone son de aplicación universal.

- **Testing estructural o de caja blanca.** Testear un software siguiendo esta estrategia implica que se tiene en cuenta la estructura del código fuente del programa para seleccionar casos de prueba—es decir, el testing está guiado fundamentalmente por la existencia de sentencias tipo *if, case, while,* etc. En muchas ocasiones se pone tanto énfasis en la estructura del código que se ignora la especificación del programa, convirtiendo al testing en una tarea un tanto desprolija e inconsistente.

  Como los casos de prueba se calculan de acuerdo a la estructura del código, no es posible generarlos sino hasta que el programa ha sido terminado. Peor aun, debido a errores o cambios en las estructuras de datos o algoritmos—aun sin que haya cambiado la especificación—puede ser necesario volver a calcular todos los casos. Por estas razones preferimos la otra estrategia de testing, aunque estudiaremos el testing estructural con cierto detalle.

  Sin embargo es interesante remarcar la racionalidad detrás de esta estrategia: no se puede encontrar un error si no se ejecuta la línea de código donde se encuentra ese error. Aun así ejecutar una línea de código con algunos casos de prueba no garantiza encontrar un posible error.

  Se dice que el testing estructural prueba *lo que el programa hace* y no *lo que se supone que debe hacer*.

- **Testing basado en modelos o de caja negra.** Testear una pieza de software como una caja negra significa ejecutar el software sin considerar ningún detalle sobre cómo fue implementado. Esta estrategia se basa en seleccionar los casos de prueba analizando la especificación o modelo del programa, en lugar de su implementación.

  Algunos autores consideran que el testing basado en modelos (**MBT**) es la automatización del testing de caja negra. Para lograr esta automatización el MBT requiere que los modelos sean formales dado que esta es la única forma que permite realizar múltiples análisis mecánicos sobre el texto de los modelos. Por el contrario, el testing de caja negra tradicional calcula los casos de prueba partiendo del documento de requerimientos

Creemos que es muy importante remarcar que estas estrategias no son opuestas sino complementarias. En nuestra opinión el testing debería estar guiado fundamentalmente por técnicas de MBT pero complementadas con herramientas de análisis de cubrimiento de sentencias de forma tal que los casos generados mediante MBT cubran al menos todas las líneas de código.

10.2 Testing Estructural

En el testing estructural los casos de prueba se seleccionan según la estructura del código fuente del programa que se está testeando.

Los conjuntos de prueba que se seleccionan siguiendo técnicas de testing estructural deben cumplir con lo que se denomina *criterio de selección de casos de prueba*. Un criterio de selección de casos de prueba, o simplemente criterio de prueba o criterio de selección, es un subconjunto de $\mathbb{F}\ ID$ donde ID es el dominio de entrada del programa que se está testeando.

Si C es un criterio de selección y T es un conjunto de prueba que pertenece a C, se dice que T satisface C. Se espera que los criterios de selección sean *consistentes*. Un criterio de selección C es consistente sí y solo sí para cualesquiera conjuntos de prueba T_1 y T_2 que satisfacen C, T_1 no encuentra un error sí y solo sí T_2 tampoco lo hace. De esta forma el tester debe elegir un único conjunto de prueba de un criterio puesto que todos los otros darán los mismos resultados.

El testing estructural puede dividirse según cómo se definen los criterios de selección.

- **Testing estructural basado en el flujo de control.** En este caso los criterios de selección se definen en base a reglas que sobre el flujo de control de los programas, y en particular se tienen en cuenta los valores de verdad de las condiciones usadas en sentencias condicionales e iterativas. En otras palabras, un criterio de selección exige que los conjuntos de prueba que lo satisfacen recorran el programa siguiendo el flujo de control de una forma específica y haciendo que las condiciones asuman valores de verdad específicos.

- **Testing estructural basado en el flujo de datos.** En esta forma de testing estructural los criterios se definen de forma tal que cumplan con ciertas reglas que gobiernan el flujo de datos de los programas. No estudiaremos criterios basados en el flujo de datos pues, si bien son más poderosos que los basados en flujo de control, el cálculo de conjuntos de prueba es muy laborioso.

En ambos casos es posible definir un orden parcial entre los criterios de forma tal que quedan ordenados según la cantidad de casos de prueba que exigen sean ejecutados. Por consiguiente los testers pueden elegir el criterio que mejor se adapte al presupuesto o tiempo disponible.

10.2.1 Grafo de flujo de control de un programa

Los criterios de selección basados en flujo de control se definen en base a lo que se denomina el Grafo o Diagrama de Flujo de Control (CFG o CFD) del programa.

Asumamos que el lenguaje de programación con el cual se escriben los programas a testear tiene la siguiente gramática:

BasicSentence ::=
skip
| *var* := *Expr*
| call($arg_1, \ldots, arg_n$)

ConditionalSentence ::=
if *Cond* then *Program* fi
| if *Cond* then *Program* else *Program* fi
| while *Cond* do *Program*done

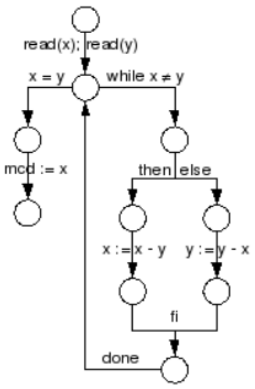
Sentence ::= *BasicSentence* | *ConditionalSentence*

Program ::= *Sentence* | *Program* ; *Program*

En ese caso el CFG de cualquier programa se construye inductivamente, teniendo en cuenta que en los CFG las flechas representan sentencias y los nodos los puntos de entrada y salida de la sentencia respectiva. Por ejemplo:

```
read(x); read(y);
while x ≠ y do
  if x > y then
    x := x − y;
  else
    y := y − x;
  fi
done
mcd := x;
```

(a)



(b)

10.2.2 Criterio de cubrimiento de sentencias

En realidad la definición de este criterio no requiere analizar el CFG debido a que es muy simple.

Seleccionar un conjunto de prueba T tal que, al ejecutar P para cada d en T, cada sentencia básica de P es ejecutada al menos una vez.

10.2.3 Criterio de cubrimiento de flecha

Este criterio se enuncia de la siguiente manera:

Seleccionar un conjunto de prueba T tal que, al ejecutar P para cada d en T, cada flecha del CFG de P es atravesada al menos una vez.

Importante. En este criterio y en todos los que siguen se debe tener en cuenta que el arco que sale de un bucle cuando la condición es falsa debe ser recorrido al menos una vez sin haber recorrido el interior del bucle en la misma ejecución. En otras palabras es necesario un caso que no lleve el flujo de control al interior del bucle.

10.2.4 Criterio de cubrimiento de condiciones

El criterio de cubrimiento de condiciones se enuncia de la siguiente forma:

Seleccionar un conjunto de prueba T tal que, al ejecutar P para cada d en T, cada flecha del CFG de P es atravesada al menos una vez y las proposiciones simples que forman condiciones toman los dos valores de verdad.

Importante. En este criterio y en todos los que siguen se debe tener en cuenta que la condición que gobierna a un bucle debe ser falsa de todas las formas indicadas por cada criterio sin antes haber sido verdadera (en el momento de evaluarla) en la misma ejecución.

10.2.5 Criterio de cubrimiento de caminos

Este criterio en general es impracticable pero se lo suele incluir en las presentaciones por una simple cuestión de completitud teórica.

Seleccionar un conjunto de prueba T tal que, al ejecutar P para cada d en T, se recorren todos los caminos completos posibles del CFG P.

Dado que es impracticable testear la mayoría de los programas siguiendo este criterio, se lo debe tener como una referencia para tratar de cubrir los caminos más críticos. La heurística mínima que debe seguirse cuando aparecen bucles es la siguiente:

- Iterar sobre cada bucle cero veces.
- Iterar sobre cada bucle el máximo número de veces posible.
- Iterar sobre cada bucle un número promedio de veces.

10.2.6 Otros criterios

Sea $\{C_i\}$ el conjunto de todas las condiciones lógicas usadas en un programa P para gobernar el flujo de ejecución. El criterio de cubrimiento de flechas requiere que el conjunto de prueba $\{d_j\}$ deber ser tal que cada C_i sea falsa y verdadera para algún d_j al menos una vez.

- Criterio de asignación de valores de verdad:

  Sea $t = \langle tr_1, ..., tr_n \rangle$ una asignación de valores de verdad a las condiciones $\{C_i\}$, es decir $\langle tr_1, ..., tr_n \rangle$ es un vector n-dimensional de valores booleanos que será asignado a cada condición de P. En otras palabras t relaciona las condiciones (compuestas) de cada estructura de control de P (y no las condiciones de una misma estructura de control). Para cada una de estas asignaciones, sea D_t el subconjunto de D que hace todas las C_i sean verdaderas o falsas según la asignación.

- Criterio de condiciones múltiples:

  El criterio de cubrimiento de condiciones múltiples puede definirse como sigue: cada conjunto de prueba debe hacer todas las condiciones C_i verdaderas o falsas de todas las formas posibles, basándose en los valores de las proposiciones simples que las componen. Por ejemplo, si C_5 es c_{51} **and** c_{52}, entonces debemos generar cuatro casos de prueba que hacen a c_{51} verdadera, c_{52} verdadera, c_{51} verdadera, c_{51} falsa, etc.

10.3 Testing en Z

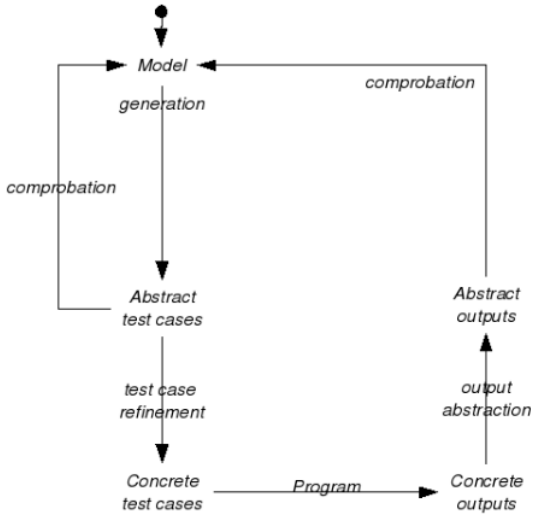


Figura 1: El proceso de testing basado en especificaciones formales.

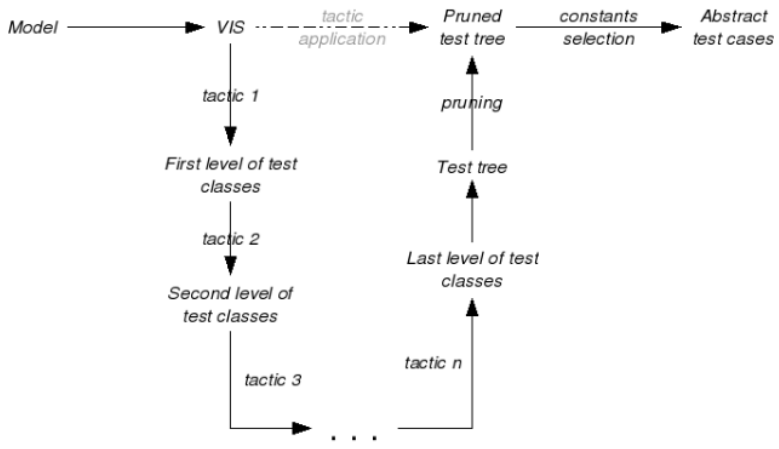


Figura 2: El proceso de generación de casos de prueba abstractos.

Tener en cuenta que falta más data, específicamente sobre Tácticas de testing.

11 Referencias

1. Apuntes de la materia. fceia.unr.edu.ar/ingsoft